# A methodology for the design and development of an aerospace-specific data repository to support data-driven research

***Une méthodologie pour la conception et le développement d'un référentiel de données spécifiques à l'aérospatiale pour soutenir la recherche axée sur les données***

**A Thesis Submitted to the Division of Graduate Studies**

**of the Royal Military College of Canada**

**by**

**Angelina X. Cui**

**In Partial Fulfillment of the Requirements for the Degree of**

**Master of Applied Science in Aeronautical Engineering**

**January 2023**

**©This thesis may be used within the Department of National Defence but copyright for open publication remains the property of the author.**

# Acknowledgments

I would like to express my deepest gratitude to my patient and supportive supervisor, Dr. Catharine Marsden, who presented me with a new world and I am grateful for the guidance and the inspiration she has given me along the way. I would also like to thank Nicolas Vincent-Boulay, who was a reliable research partner and always to my assistance when I need help in this project; and to everyone in my research group for the support they have provided and the pleasant atmosphere they have created. I am also grateful to my family for having my back all the time.

Thank you to NSERC (Natural Sciences and Engineering Research Council of Canada) and the NSERC Chair in Aerospace Design Engineering (NCADE) industrial partners for their financial support, and to Royal Military College of Canada.

# Abstract

A data-driven approach to aerospace research or applications could be beneficial in the era of big data, but accessibility and usability of the data make this challenging. There have been attempts by government and academic researchers to resolve this issue by creating large generic databases. However, the issue remains since those databases may not be able to satisfy researchers' needs. In this thesis, a systematic approach is presented that can be used to create a compact, efficient, and aerospace application-specific database based on specific research requirements. The methodology, which utilizes aviation-related data that are publicly accessible, follows the data ETL (Extract, Transform, and Load) approach based on domain-specific requirements. A data visualization tool was created to help researchers better understand the data. The approach was tested with a case-study simulated environment designed for the purpose of investigating interactions between entities in a shared airspace. As a result of the implementation, a relational database containing real-life archived air traffic and weather radar data was created, as well as a map-based data visualization tool. The result is verified with test cases to demonstrate the flexibility of the approach for creating different true-to-life flight scenarios.

# Résumé

Une approche de recherche axée sur les données pourrait être bénéfique à l'ère du mégadonnées pour des applications aérospatiales, mais l'accessibilité et la facilité d'utilisation de ces données en font un défi. Des gouvernements et chercheurs universitaires ont tenté de résoudre ce problème en créant de grandes bases de données génériques; cependant, le problème demeure puisque ces bases de données peuvent ne pas satisfaire les besoins des chercheurs. Dans cette thèse, on présente une approche systématique qui peut être utilisée pour créer une base de données compacte, efficace, et spécifique à une application aérospatiale, basée sur des exigences de recherche spécifiques. La méthodologie, qui utilise des données relatives à l'aviation accessibles au public suit l'approche ETL (extraction, transformation et chargement) des données en fonction des exigences spécifiques au domaine. Un outil de visualisation des données a été créé pour aider les chercheurs à mieux comprendre les données. L'approche a été testée avec un environnement simulé conçu pour étudier les interactions entre des entités dans un espace aérien partagé. L'implémentation de cette approche a permis de créer une base de données relationnelle contenant des données sur le trafic aérien, des données de radar météorologique archivées, ainsi qu'un outil cartographique de visualisation des données. Le résultat est vérifié avec des cas d'essai afin de démontrer la flexibilité de l'approche pour créer différents scénarios de vol réalistes.

# Table of contents

**Abstract** .......... **iii**

**Résumé** .......... **iii**

**Table of contents** .......... **iv**

**List of Tables** .......... **vii**

**List of Figures** .......... **viii**

**List of Acronyms** .......... **ix**

**1. Introduction** .......... **1**

**2. Literature Review** .......... **3**

2.1 Background information - big data .......... 3

2.2 Use cases in other academic fields .......... 4

2.3 Challenges associated with big data implementation .......... 5

2.3.1 Data visualization .......... 5

2.3.2 High volume, high speed, and data sources .......... 6

2.3.3 Data management .......... 7

2.4 Database and database management system .......... 8

2.4.1 Relational and non-relational database types .......... 8

2.4.2 DBMS performance .......... 10

2.5 Data-driven research in the aviation industry .......... 11

2.5.1 Historical big data and aerospace simulation research .......... 13

2.6. Research-specific aerospace data collections .......... 16

2.7 Research objective .......... 20

**3. Research Methodology** .......... **21**

3.1. Data source selection .......... 21

3.1.1 Weather data ........................................ 22
3.1.2 Air traffic data ........................................ 26
3.2 Independent data repository ........................................ 28
3.3 Data ETL ........................................ 30
3.3.1 Data Extraction ........................................ 30
3.3.2 Data transformation ........................................ 37
3.3.3 Loading and data storage ........................................ 40
3.4 Data visualization and user interaction ........................................ 41
3.5 Process verification using a proof-of-concept test case ........................................ 43
**4. Case study implementation ........................................ 45**
4.1 Data source selection ........................................ 46
4.2 Database structure design ........................................ 47
4.3 Data ETL ........................................ 49
4.3.1 Data Extraction ........................................ 49
4.3.2 Data transformation ........................................ 55
4.3.3 Loading data to storage ........................................ 60
4.4 GUI development for data visualization ........................................ 61
4.5 Verification with test case ........................................ 68
**5. Conclusion ........................................ 73**
**6. References ........................................ 75**
**Appendix A: NEXRAD data extraction ........................................ 83**
**Appendix B: ADS-B data extraction ........................................ 85**
Appendix B1: ADS-B data extraction with Python wrapper ........................................ 85
Appendix B2: ADS-B data extraction with custom Python script ........................................ 86
**Appendix C: Weather radar data processing ........................................ 89**

**Appendix D: Loading data to the database for storage** .......... **92**
**Appendix E: Prototype #1** .......... **100**
**Appendix F: Prototype #2** .......... **103**
Appendix F1: Graphical user interface .......... 103
Appendix F2: Design the main map .......... 111
Appendix F3: Obtain airspace and weather data from data files .......... 117
Appendix F4: Obtain NAVID chart from data files .......... 121
Appendix F5: Obtain elevation profile from data files .......... 122
**Appendix G: Prototype #3** .......... **124**
Appendix G1: Design the map layout .......... 124
Appendix G2: Main execution file .......... 128
Appendix G3: Convert data in CSV file to arrays .......... 144
Appendix G4: Mapping ADS-B data for display .......... 148
Appendix G5: Mapping NexRad data for display .......... 155

# List of Tables

Table 1. Comparison of SQL, NoSQL and NewSQL ........ 10
Table 2.Information about different weather data ........ 24
Table 3. Information available in ASDI data stream ........ 26
Table 4. AWS CLI command syntax ........ 34
Table 5. SQL query statement ........ 42
Table 6. Dynamic entity and Static entity ........ 46
Table 7. ADS-B data content from the OpenSky Network ........ 51
Table 8. Raw weather radar data (part 1) ........ 56
Table 9. Raw weather radar data (part 2) ........ 56
Table 10. Data content of weather radar dataset ........ 57
Table 11. Data content of ADS-B dataset (part1) ........ 57
Table 12. Data content of ADS-B dataset (part2) ........ 58
Table 13. Standardized weather radar data ........ 58
Table 14. Standardized ADS-B data (part 1) ........ 58
Table 15. Standardized ADS-B data (part 2) ........ 59
Table 16. ADS-B data file size comparison ........ 59
Table 17. Weather data file size comparison ........ 60

# List of Figures

Figure 1.Data ETL process ........ 8
Figure 2. Workflow ........ 21
Figure 3. The service available in NOAA ........ 22
Figure 4. Structured data vs unstructured data ........ 29
Figure 5. Direct download NEXRAD data from NCEI ........ 31
Figure 6. Amazon S3 structure ........ 33
Figure 7. AWS CLI initial configuration ........ 33
Figure 8. Accessing data from Amazon S3 using AWS CLI ........ 35
Figure 9. Accessing data using Impala Shell ........ 36
Figure 10. Decoding raw radar data file with WCT ........ 38
Figure 11. Data transformation process ........ 38
Figure 12. Data standardization with two raw datasets ........ 39
Figure 13. Load cleaned data to storage ........ 41
Figure 14. Effect on tasks due to change of decision from "research requirement" to "data format" .. 44
Figure 15.Database schema ........ 48
Figure 16.NCEI Amazon S3 bucket ........ 50
Figure 17.Downloading weather data from two different days ........ 51
Figure 18. ADS-B data download log comparison ........ 53
Figure 19. ADS-B data file comparison ........ 54
Figure 20. An overview of database ........ 61
Figure 21. ADS-B data representation in Prototype #1 ........ 62
Figure 22. Interface of Prototype #2 ........ 63
Figure 23. Data layers in Prototype #3 ........ 65
Figure 24. The interface of Prototype #3 ('Flight' map style) ........ 66
Figure 25. The interface of Prototype #3 ('Satellite' map style) ........ 68
Figure 26. Introduce novel air vehicle to Prototype #3 (bird's eye view) ........ 69
Figure 27. Introduce novel air vehicle to Prototype #3(3D view) ........ 70
Figure 28. Adding new data layer to Prototype #3 ........ 71
Figure 29. Generate novel simulate airspace environment ........ 72

# List of Acronyms

| Acronym | Definition |
|---|---|
| ACES | Airspace Concept Evaluation System |
| ACID | Atomicity, Consistency, Isolation, and Durability |
| ADS-B | Automatic Dependent Surveillance Broadcast |
| Amazon S3 | Amazon Simple Storage Service |
| ASDI | Aircraft Situation Display to Industry |
| ATM | Air Traffic Management |
| AWS | Amazon Web Services |
| CLI | Command Line Interface |
| CSV | Comma-separated values |
| DBMS | Database Management systems |
| DOD | United States Department of Defense |
| ETL | Extract, Transform, and Load |
| FAA | Federal Aviaiton Administration |
| FACET | Future ATM Concept and Evaluation Tool |
| GUI | Graphical User Interface |
| IoT | Internet of Things |
| NAS | National Airspace System |
| NASA | National Aeronautics and Space Administration |
| NCEI | National Center for Environmental Information |
| NEXRAD | Next Generation Radar |
| NOAA | National Oceanic and Atmospheric Administration |
| NOAA’s WCT | NOAA's Weather and Climate Toolkit |
| NWS | National Weather Service |
| SQL | Structured Query Languag |

# 1. Introduction

The availability of big data is driving several recent advances in aircraft-related research, for example aircraft trajectory optimization [1] and prediction [2]; flight delay prediction [3], [4]; and the development of simulated environments [5]–[7] to support research projects [2], [8]–[16]. But progress is slowed because of challenges associated with accessing and processing the data so that it is practically useful to aerospace researchers.

Big data is a term that is used to describe data generated at high speeds and in large volumes from various sources and in differing formats. Big data is increasingly available, and data-driven research is gaining traction across multiple industries, including aerospace. Challenges to discipline-specific, data-driven aerospace research include the accessing, processing, and management of large amounts of both raw and transformed data.

Aviation-related data-driven research is increasingly popular, and real-life historical aviation data is being used to support several research projects including the development of aircraft trajectory optimization algorithms and the study of the relationship between weather and air traffic ground delay programs. The difficulties associated with acquiring and storing the relevant data can be found in the literature, and the creation of aviation databases has become a research topic in itself.

The literature presents examples of both governmental and academic researchers creating large generic databases to store pre-processed data meant to serve a broad range of aviation research projects. However, the data stored in these generic databases can differ from what the researchers need, and the pre-processed data may still need to be converted into another format to meet the needs of specific projects. One solution is to create smaller, more targeted, independent database systems designed for specific applications, where data is selected and processed based on carefully defined research requirements before being stored in the database.

This thesis presents a methodology for creating such a targeted database to meet specific research requirements. The methodology employs the data ETL (Extract, Transform, and Load) processes, with an additional phase for creating a data visualization tool meant to make the data more understandable to researchers. In this thesis, a proof-of-concept database is created to support a project investigating interactions between entities in a shared airspace. The result of the implementation is a relational database that contains processed weather radar and air traffic data,

as well as a map-based data visualization tool. The methodology presented can be adapted to a variety of mid-project changes to the given research requirements, and researchers can use the processed data not only to recreate real-life airspace environments as they happened in the past but also to combine historical scenarios and/or introduce novel entities into the simulated real-life airspace environment.

The thesis is organized into chapters, with Chapter 1 being the introduction. Chapter 2 presents the literature review related to the research. Chapter 3 presents the approach for the design and development of an aviation-specific data repository to support data-driven application. Chapter 4 presents the implementation and validation of the approach through a case study. Chapter 5 concludes and recommends potential uses for the contributions of this research to future research projects.

# 2. Literature Review

This chapter presents the literature review conducted related to the research, beginning with the background information on big data in Section 2.1. Section 2.2 presents several use cases of big data in other academic fields, and Section 2.3 outlines some common challenges associated with big data implementation faced by researchers. In Section 2.4, the types of database and database management systems are introduced. Section 2.5 provides some examples of data-driven research conducted by other researchers in the aviation industry, and Section 2.6 introduces a selection of data warehouses developed as research-specific aerospace data collections. Chapter 2 is concluded with a description of the research objective in Section 2.7.

## 2.1 Background information - big data

In the era of big data, individuals and organizations can generate, collect and exchange large amounts of digital data every day through the Internet of Things (IoT) [17]. The IoT refers to a network of physical devices or software connected through the internet, and the purpose of this network is to collect and exchange data with other devices. The data collected by the IoT can be imagined as being from a very large number and variety of data sources that keep feeding information to a big data pool, which leads to the term "big data".

There is currently no standard definition of the term "big data" although various definitions can be found. The National Institute of Standards and Technology (NIST) is engaged in an ongoing effort to determine a precise definition, and in their latest publication stated that "Big Data refers to the need to parallelize the data handling in data-intensive applications." [18]. The core characteristics of big data can be defined by the 3 Vs; volume, velocity, and variety [19], [20]. Volume refers to the size of the data generated and stored, where the size of big data is usually larger than terabytes and cannot be easily processed with a normal computer. Variety describes the diverse types, sources, and formats of the collected data, where, because of the IoT, different physical devices are used to collect data resulting in a variety of data formats. Velocity reflects the speed of data generation and processing.

## 2.2 Use cases in other academic fields

Commercial enterprises see the large amount of available data as a major economic opportunity with the potential to help them make smarter business decisions [21], whereas academic researchers see great potential in how big data analytics could impact science, technology, and other fields [22]. Gudivada et al. [17] believe that big data is enabling new directions for scientific research that was previously limited by the volume of available data. Applications for the use of big data exist across many fields.

Bai and Bai [23] propose the idea of using sport-related big data analysis to help coaches and athletes know more about themselves and adjust their training activities. Sports-related big data analysis can also be used to analyse the behaviour of competing teams, making it possible for the coach and athletes to plan strategies to take advantage of their opponents' weaknesses. Finally, big data analysis can also be used to identify rising stars in the sport.

Han et al. [24] discuss the development of a platform to collect, store and analyse information related to Android applications. The platform can be used to identify the common characteristics of malicious applications and can alert users if such characteristics exist in a newly added application.

Chen et al. [25] propose using big data to develop an agricultural decision model to achieve precision agriculture. The purpose of precision agriculture is to obtain a more abundant harvest with less resource consumption to optimize the agricultural economy. The research group is currently applying the model to a real-world experiment on a banana field.

Munshi et al. [26] propose the design and implementation of a big data platform for educational analytics. With the massive amounts of available educational data, models can be built to predict incidents such as student drop-out rates, or to recommend specific courses for individual students. The authors believe that the platform can provide academic advisors valuable information allowing them to give more attention to students who might need their help, and ultimately enhancing education quality.

Big data technology is also being used to predict market trends, identify target customers, and develop products that better fit the consumer's needs. Xiong [27] believes that big data marketing is better than traditional marketing in the automobile industry because it provides insights into

consumer demand and identifies consumers' interests, thereby allowing enterprises to produce automobiles that better meet the needs of consumers. Su [28] proposes the idea of using data mining and cloud computing technology to analyse data from the tourism industry. The author believes this will help develop innovative management models which can provide personalized marketing plans for tourism users. Zhe et al. [29] have designed a luxury brand marketing service model using big data. The authors believe their model will provide the possibility for precision marketing and provide consumers with exclusive services.

Gupta and Rani [30] have conducted a literature review on both academic and industry publications with respect to big data published from 2000 to 2017. Through the bibliometric study, they find that the common research challenges preventing researchers from realizing the value of big data include high data volume, data format and data source diversity, data sets correlation, and data visualization.

## 2.3 Challenges associated with big data implementation

### 2.3.1 Data visualization

Identifying a relevant and understandable way to visually present data is a major challenge associated with big data implementation. Data visualization satisfies the 'visual need' of the human mind and can help humans to understand meaningful information obtained from a large quantity of data regardless of the different cultural backgrounds or the spoken language of the reader [31]. Data visualization can be used to provide insight into information and help the reader to understand and form an opinion on a complex context in a "storytelling" way [32].

Mani and Fei [33] believe that data visualization has an essential role in big data analytics. Big data analytics refers to the process of applying analytical techniques to data to discover reliable facts or potential information. However, the authors state that the complexity of big data can affect the efficiency of a data visualization tool. They believe that this problem can be solved if the amount of data is reduced, for example when only the data requested by the user can be extracted and processed to create a visualization.

Agrawal et al. [34] discuss the importance of data visualization tools for helping people understand collections of data and to support real-time decision making. The research group believes that it is easier for people to understand data in a graphical manner. They point out that there are many

challenges associated with real-time big data visualization, including extracting data from data sources, determining the essential information to be displayed, and using the data to create an effective graphical image. The strategy proposed by the group is data reduction, which is reducing the big data to smaller manageable data before processing it into visualization.

Eldin et al. [35] believe that data collected by the IoT needs to be pre-processed and presented in a meaningful way before it can be used by researchers. They believe that data visualization is a simple and fast way to deliver messages and represent complicated things because it summarizes large amounts of data using a graphical interface. To convert data into a graphical format requires the following steps: data extraction and data fusion; identification of a suitable graph type or visualization model based on the user's objective; and the generation of a visual representation capable of delivering meaningful messages to users. The challenge associated with this process is to summarize the data for a specific domain in order to create appropriate visuals to support the needs of the user.

The literature search shows that data visualization is an effective tool to help humans from different cultural and technical backgrounds understand complex information in a ‘storytelling’ way [31], [32]. But the complexity of big data makes big data visualization challenging. Data needs to be processed before converting to visualization because applying a very large amount of data to a visualization system is inefficient, particularly in the case of real-time implementation [34]. A common strategy proposed by researchers to solve this challenge is the reduction of the amount of input data before sending specific data related to the user’s interest to the visualization system [33]–[35].

### 2.3.2 High volume, high speed, and data sources

Digital data is constantly being created, and the size of the data generated is growing rapidly as data collection technology becomes more advanced. In 2012, IBM estimated that there were 2.5 exabytes ($10^{18}$) of data generated every day and that the amount was expected to grow [36]. The hard drive capacity of a laptop can be range from 160 gigabytes ($10^{9}$) to more than 2 terabyte ($10^{12}$) [37], which means it would require at least 1.3 million laptops to store all the data. The International Data Corporation (IDC) predicted in 2011 that the size of the data generated every day will grow 50 times by the year 2020, and that the growth will be driven by the physical devices or software embedded in systems that continually connect data[38].

In 2015, the Australian Square Kilometre Array Pathfinder (ASKAP) demonstrated that they are presently able to obtain 7.5 terabytes ($10^{12}$) of astronomical data per second and they expect the number to be 100 times bigger by 2025 [39].

The IoT is generating vast amounts of data at a high speed, but the data cannot be directly used in many applications because the dataset is too large and not processed. Data collected by different devices often exist in a variety of formats, even for the same type of data. There is a requirement to reduce the size of the dataset and to unify the raw data format before these data can be useful to the broader research community.

### 2.3.3 Data management

Data management refers to the action of managing data as a resource. Effective data management not only requires having a reliable method to access, integrate, clean, store and prepare data for other applications, but also requires robust techniques for maintaining the data storage system and ensuring data security [40].

Data migration is a specific data management operation that refers to the action of transferring data from its origin to a new data storage place for an identified application [41]. Data migration can be divided into three sub-processes identified as Extract, Transform and Load. Data ETL is an important step in big data analysis, and a successful data ETL provides researchers with a clean data warehouse to use [42]–[45]. “Extract” is the process of extracting data in different formats from various data sources. "Transform" is the most complicated part of the ETL process and requires the application of transformation techniques to clean the raw data. Transformation techniques include cleansing, filtering, and restructuring the raw data, and converting it to the desired format [45]. The transformation process is case-specific; it requires manual analysis of the datasets, and a clear definition of the intended use of the data before defining the appropriate structure for the data warehouse. Once the clean data is stored in the selected data warehouse it is ready to be used in the chosen application. Figure 1 shows the conceptual model of the data ETL process.

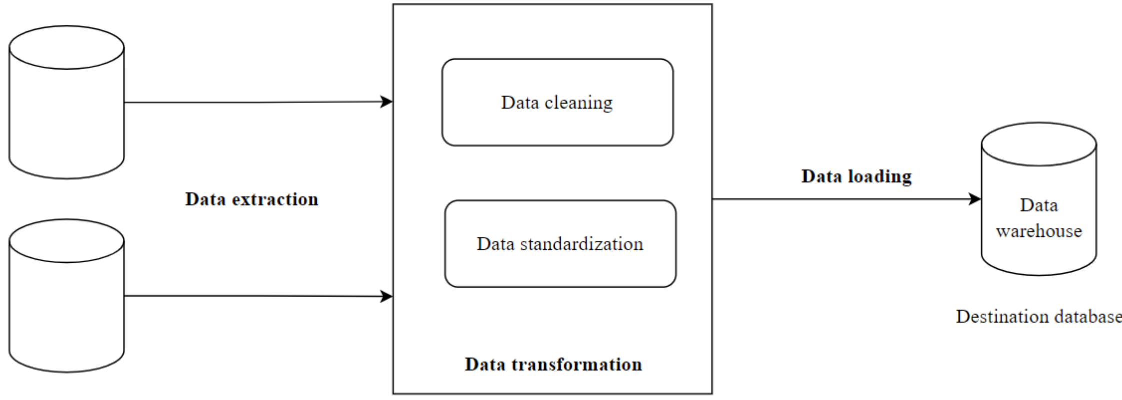


*Figure 1.Data ETL process (Adapted from 'Design of ETL Tool for Structured Data Based on Data Warehouse', by J. Wang et al.[44]*)

A well-designed data ETL process extracts data from multiple data sources, enforces data types and standards, and ensures structural compliance with the output requirements determined by the application. This is a key process as it will bring different data together in a standard, homogeneous environment as a data warehouse.

## 2.4 Database and database management system

The term "database" refers to a collection of data that are related to each other. In the physical world, a database can be a series of books or papers. In computing, a database is a collection of organized data that is stored and can be accessed electronically in a computer system. Software that is used to manage, maintain, or interact with databases is called a database management system (DBMS). The DBMS serves as an interface between the database and the end-users or software applications and ensures the required data is organized and can be easily accessed. There are different types of DBMS depending on the database type they are designed to interact with.

### 2.4.1 Relational and non-relational database types

The concept of the relational database is based on a model proposed by E. F. Codd in 1970 [46]. In a relational database, there exists a database structure schema that describes relationships between each of the parameters stored in the database. The schema should be defined before the

actual database is created and very few modifications can be made once it is defined. In the operational aspect, relational databases have Atomicity, Consistency, Isolation, and Durability (ACID) properties. When a relational database is used, each transaction is an atomic operation; if one part of the system fails, the entire system fails. The isolation and consistency of a relational database are ensured by each transaction behaving independently and being subject to a set of rules. Finally, relational data is durable in the sense that if someone changes the database, other parties who have access to it will be able to see the same changes [47]. The ACID properties ensure high data consistency between the database and the end-users. These characteristics of the relational database make it an excellent system for banking or financial systems. The Structured Query Language (SQL) is a programming language designed specifically for managing and querying relational databases in a DBMS, thus a relational database is also called an SQL database.

The NoSQL database (non-SQL or non-relational database) is a more recently developed type of storage method that is becoming increasingly popular. In contrast to relational databases, NoSQL databases do not require a predefined schema; they store data regardless of its structure and content, allowing greater flexibility [47]. NoSQL databases can store unstructured or semi-structured data such as video clips, digital figures, or document files. Large amounts of such data is being generated every day in web and mobile applications, and there is a need for a storage method that can collect and store these varied formats of data with low latency, which is a difficult task for a relational database [48]. Unlike relational databases, NoSQL databases are faster at processing data because they do not need to adhere to ACID properties [49], [50].

There is a trade-off between performance speed and database complexity [50]. Therefore, one main concern with a NoSQL database is how to ensure its reliability and consistency. Another difficulty is that NoSQL does not have a well-defined query language, making complex data query difficult.

Scalability is another difference between SQL and NoSQL databases [47], [51]. An SQL database is typically hosted on a single server and can be scaled vertically. The scalability of a SQL database is achieved by adding additional memory, processors, and storage to the server. NoSQL databases, on the other hand, are horizontally scalable and are often designed to work across cloud servers.

A third database type is NewSQL, which can be considered an improved version of the relational database, which still maintains the ACID properties but can have the scalable performance of a NoSQL database [52]–[54]. Table 1 shows a high-level comparison of the SQL, NoSQL, and NewSQL database types.

*Table 1. Comparison of SQL, NoSQL and NewSQL*

| **Characteristic** | **SQL** | **NoSQL** | **NewSQL** |
|---|---|---|---|
| ACID properties | Yes | No | Yes |
| Relational | Yes | No | Yes |
| Support unstructured data (e.g.: video, audio, etc.) | No | Yes | In some cases |
| Standard query language | Yes | No | Yes |
| Community support | Very high | High | Low |
| Scalability | Vertically scalable | Horizontally scalable | Horizontally scalable |

### 2.4.2 DBMS performance

There are several DBMS available for managing the three types of databases. Research has been conducted comparing the performance of different DBMSs, and results show that DBMS for NoSQL databases is not always more efficient than DBMS for relational databases[49], [55], [56]. The performance of the DBMS depends on the database operation being performed.

Li et al. [55] conducted performance comparison on several NoSQL DBMSs against Microsoft SQL Express. MongoDB, Hypertable, Apache CouchDB, Apache Cassandra, RavenDB, and Couchbase are the NoSQL DBMS chosen for the experiment. The research group found that the NoSQL databases were not always outperforming the SQL database, and that the efficiency of the database varied depending on the operation. For reading, writing, and deleting operations the research study found that only MongoDB and Couchbase consistently outperformed the Microsoft SQL Express.

Fatima et al. [49] also conducted research on the performance of three different types of DBMSs. In their experiment, MongoDB was chosen as the NoSQL DBMS, MySQL was chosen as the SQL DBMS, and VoltDB as the NewSQL DBMS. The research group used these three DBMSs to store and manage data and compared their performance. According to their research, VoltDB always

outperformed MongoDB and MySQL in terms of both read and write operations. As for the other two DBMSs, MySQL was more efficient for the read operation while MongoDB was more efficient for the write operation.

Rautmare et al. [56] conducted similar research to compare the performance of the read and write operation for MySQL and MongoDB. The results show that MongoDB outperformed MySQL in some cases, but the response time in MySQL was more stable. The authors state that choosing a DBMS should depend on the requirements associated with the database application.

## 2.5 Data-driven research in the aviation industry

Data-driven research projects related to aviation are relatively common in the literature. Li and Ryerson [57] have conducted a literature review of 200 aviation data-driven publications published after 2010. The authors find that the literature references a variety of data types, and that the same data may come from different sources. They identified 16 data categories and found that each one of them had at least 5 available sources. They point out that, because of the existence of different data sources for the same data, the nomenclature used to describe the information can be inconsistent, making the data difficult to access and problematic to use. With respect to data availability, the researchers found that 24% of the 200 publications use publicly available data, whereas the rest rely on proprietary sources. The authors believe that standardizing data source nomenclature in the industry and increasing the amount of publicly available data would greatly benefit aviation data-driven research. A selection from the literature reviewed by Li and Ryerson is provided in the rest of this section.

Ben Abda et al. [58] examined domestic origin-destination traffic and fares at America's 200 largest airports from 1990 through 2008, focusing on the effects of the arrival and growth of low-cost carriers by analysing air carrier data obtained from the US Department of Transportation Bureau of Transportation Statistics.

Alderighi and Gaggero [4] used flight schedule and meteorological data to study the flight cancellation rate and show that airlines belonging to global alliances are more likely to cancel a flight under the same weather condition.

Fu and Kim [59] studied the relationship between airport passenger leakage at small local airports and the relationship with multiple explanatory factors such as travel group size and airfare. The

authors employ 8 years of publicly available data in an econometric model including airport passenger traffic, airline services, driving distance between airports, census information, and aviation fuel costs.

Cadarso et al. [60] studied the competition between airlines (legacy and low-cost) and high-speed rail. The authors developed a model for generating airline schedules and used real operational data obtained from a Spanish airline company as input to validate their model. The results they obtained from the validation shows that their model generated similar flight schedules to the actual flight schedule used by the airline company. The validated model was then used to predict the impact on airline scheduling caused by the entry of high-speed rail into the transport market.

Ren et al. [3] applied machine learning techniques to study the relationship between weather and Ground Delay Programs (GDP) issued at Newark Liberty International Airport from 2010 through 2014. The authors take GDP advisory data, FAA flight data, and forecast and observed weather data from U.S. National weather services to create a merged master data repository to support their research.

Ng et al. [1] developed a trajectory optimization algorithm that minimizes the cost of time and fuel burn, especially for cargo flights. Their algorithm was validated in a MATLAB simulation by creating a new trajectory for specific cargo flights based on air traffic data from October 2010, including wind data for the same period obtained from the National Oceanic and Atmospheric Administration.

Rakas et al. [61] developed a generalized method to evaluate the impact of equipment outages on airport throughput and the probability of a separation loss between aircraft. The research involves a variety of factors including weather conditions, aircraft type, and landing and departure times. The authors highlight the fact that the data they need to conduct their research cannot be found in a single database. Their study combined historical data from three Federal Aviation Administration (FAA) databases: the Aviation System Performance Metric (ASPM), the Remote Monitoring and Logging System (RMLS), and the Performance Data Analysis and Reporting System (PDARS).

In reviewing some of the publications that appear in Li and Ryerson's survey [57], it becomes clear that researchers use historical aviation data obtained from real life to support a variety of research projects. The industry uses real-life data as input to validate new concepts or algorithms being developed [1], [3], [60], [61], or they analyse the data and obtain some insight regarding the

messages within the data [4], [58], [59]. In a third important application, real-life historical data is used to support the development of simulation software. Section 2.5.1 introduces and reviews two such simulation software systems developed by NASA and used for research.

### 2.5.1 Historical big data and aerospace simulation research

The Future ATM Concepts and Evaluation Tool (FACET) is a simulation and analysis tool developed by National Aeronautics and Space Administration (NASA) in the late 1990s [5]. Advanced air traffic management concepts can be explored, developed, and evaluated using the simulation environment provided by FACET. FACET can create simulations and playbacks with the support of real-life historical data on air traffic, airspace constraints, aircraft performance, and weather [6], [8]. The software tool was developed to satisfy the requirements of NASA Air Traffic Management (ATM) researchers by using real-world data to create simulations. FACET has been used as a testbed for air traffic management related research in subject areas including airspace complexity, conflict detection and resolution, and flexible airspace utilization.

Sridhar et al. [8] presented a three-step hierarchical method to integrate air traffic flow management initiatives for the purpose of avoiding regions of severe weather and preventing congestion in the airspace sector. A simulated environment is created using FACET with 24-hour historical air traffic data to evaluate the method.

Bilimoria et al. [9] evaluated the performance of two Conflict Detection and Resolution (CD&R) schemes in a simulated air traffic environment provided by FACET. A 6-hour test scenario involving nearly a thousand aircraft was modelled in FACET to support the evaluation work. The initial condition of the test scenario was formed by using actual air traffic data, and the two CD&R schemes were applied to the simulation to evaluate their influence on the air traffic in terms of safety, efficiency, and stability.

Sheth et al. [10] conducted an analysis of five airspace tube structures using FACET, including three existing and two new designs. Using 24-hours of historical air traffic data, FACET was used to create a simulated environment to test the performance of the designs. The authors defined the following metrics to measure performance; the spatio-temporal utilization of the airspace, the frequency and angles at which the aircraft cross the tubes, and the separation distance between aircraft with and without tubes.

Although FACET can generate simulated environments to support Air Traffic Management related research, it cannot generate simulations of 3D aircraft trajectories or include flight delays caused by airport ground operation. As a result, NASA has developed the ACES software system that can be used to simulate aircraft trajectories with given initial conditions and performance parameters specific to an aircraft model.

The Airspace Concept Evaluation System (ACES) is a fast-time, gate-to-gate simulation and modelling tool for the National Airspace System (NAS). NASA developed ACES in 2001 and is continuously updating and incorporating new features [7]. ACES can be used for investigating current operations, future operating concepts, and new tools and architectures for the NAS. The simulated environment in ACES is created using official data published by regulatory agencies including the Rapid Update Cycle (RUC) for enroute wind, the Kinematic Trajectory Generator (KTG), and the Base of Aircraft Data (BADA) for aircraft performance. In addition to using multiple databases to create realistic simulated environments, the latest version of ACES also includes a library of plugins that can be used by researchers to support the development and evaluation of NextGen concepts.

Thipphavong et al. [2] developed an adaptive weight algorithm to improve the accuracy of aircraft climb trajectory prediction and they used ACES to establish a proof-of-concept. ACES was selected because it can generate realistic aircraft trajectories using aircraft models derived from the BADA. The algorithm adjusts the gross weight of the aircraft model based on the rate of change in kinetic and potential energy and uses the adaptive weight algorithm to predict the aircraft's climb trajectory. The researchers believe that the successful development of this algorithm will help reduce air traffic control workload; improve the automation level of separation assurance; and increase the capacity of the Next Generation Air Transportation System.

Chen et al. [11] investigated four Detect-and-Avoid (DAA) Well Clear definitions between non-cooperative aircraft and Unmanned Aircraft Systems (UAS). The authors believe that DAA systems are essential to ensure the safe integration of UAS into the NAS. The DAA Well Clear is a separation standard used in DAA systems, and it is important to correctly identify the appropriate Well Clear definition for UAS DDA systems. The research group uses ACES to generate UAS trajectories, and the simulation of the encounter scenario is conducted by pairing a UAS trajectory with a non-cooperative aircraft trajectory.

Satapathy et al. [12] evaluate the sensitivity of a new Efficient Descent Adviser (EDA) tool to predict trajectory errors. The EDA tool evaluated by the group is a trajectory-based method and its performance depends directly on the accuracy of the trajectory prediction. The research group conducted the sensitivity analysis by using ACES along with its Kinematic Trajectory Generator (KTG) to model both actual and predicted flight trajectories. The difference between the two is due to the uncertainty in the data applied in the trajectory prediction calculation. The EDA is applied to both trajectories in the simulation, allowing the researchers to evaluate the EDA performance under variations in trajectory prediction uncertainty requiring controller intervention.

Apart from being used to generate aircraft trajectories, ACES is also a useful research tool for creating realistic simulations related to airport operational metrics, for example airport throughput and flight delays. The accuracy of the output in terms of this type of simulation had been verified by Zelinski et al. [13], [14] using real-world historical flight and weather data as input to ACES. The research group obtained output including airport throughput, flight delay, and flight tracks from ACES, which they then compared to the real-world data of the reproduced day in the simulation. The results show that the output obtained from ACES is highly correlated to real-world data.

Erzberger et al. [15] presented the design of a ground system that can resolve problems such as aircraft conflict, arrival schedule, and convective weather avoidance as a means of accommodating piloted and non-piloted aircraft with reduced dependency on human controllers. ACES was employed by the research group to test their design. A 24-hour period of historical arrival and departure data for the Dallas Fort Worth International Airport and the Dallas Love Field airport is used as input to create a simulated environment in ACES.

Smith et al. [16] investigated the use of larger aircraft and alternative routing to complement the capacity benefits expected from NextGen in 2025. The research group uses ACES to access NAS delays for the 2025 demand projected by a Transportation System Analysis Model. The demand projection for 2025 is made based on real life air traffic data obtained in 2006. The results show that using larger aircraft with more seats on high-demand routes and introducing new direct routes can significantly reduce delay and complement NextGen improvements.

Although ACES can generate realistic aircraft trajectories and simulate flight delays at specific airports, the software does not include the capability of simulating weather. Instead, the only

meteorological component available in ACES is the wind effect, and this is done by adjusting the aircraft's ground speed in the simulation in accordance with the input weather data.
The examples of FACET and ACES usage described above illustrates how researchers in the aviation industry are using real-life historical data to create simulated environments as a means of testing new concepts. Both FACET and ACES require various types of input data depending on the research-specific simulations [5]–[7]. As a result of studies including those described, researchers have remarked that the quality of the real-world data can be inadequate. For example, data obtained from one data source might be incomplete and will require combining data from other data sources before using them as input to simulation software [13], [14]. As a result, aviation researchers have begun to address the problem of data complexity, accessibility, and compatibility by undertaking projects aimed at processing and grouping available archived aviation data in "data warehouses" as a means of providing clean data sources.

## 2.6. Research-specific aerospace data collections

The development of research-specific databases capable of providing relevant, clean, organized and manageable data collections is becoming an increasingly popular research topic [62], [63]. These collections are meant as data "warehouses" and do not process or apply the data to simulations or other software applications, but rather focus on gathering, cleaning and making the data accessible so that researchers may use them for specific modelling applications. Eshow et al. [62] introduced the design and implementation of a data warehouse named 'Sherlock' in support of ATM research at NASA's Ames Research Center. The purpose of Sherlock is to serve as a centralized data repository that holds all relevant ATM data and enables NASA researchers to access data for their own purposes. The author points out that the success of creating Sherlock depends on continuous access to reliable and robust data sources. The data stored in Sherlock comes primarily from the FAA and the National Oceanic and Atmospheric Administration (NOAA). The two organizations have multiple sub-departments that collect air traffic-and weather-related data that could be used to describe the NAS. An open-source software application is used to extract, transform and load data from data sources to the storage place in Sherlock. The data stored in Sherlock can be accessed through a web application, and researchers can download the data they need from the Sherlock web application instead of trying to find the data they need from the internet.

For example, Kuhn [64] proposed a methodology for characterizing historical flight days based on aviation weather and air traffic conditions in a given region to provide input for Traffic Flow

Management (TFM) decision-making. The author extracted weather and air traffic data sets from Sherlock and characterized the information on a case-by-case basis. The cases developed were employed in the analysis of the past use of TFM initiatives as a means of improving the performance of the air transportation systems and reducing air traffic management workload in similar situations.

Pang et al. [65] proposed a neural network model for weather-related aircraft trajectory prediction using raw air traffic and weather data from Sherlock for a database used to train their model. The model generated the trajectory prediction based on the aircraft's current flight plan, the history flight tracks, and the weather conditions at the time.

Evans and Lee [66] conducted research on air traffic schedule delays caused by weather or air traffic congestion. The authors applied data mining techniques to historical data extracted from Sherlock for arrival operations at the Newark Liberty International Airport between June and August 2010. The authors believe their work will contribute to a better understanding of how these factors contribute to the occurrence of schedule delays and help to improve ATM decision-making.

Sherlock is a platform for reliable ATM data collection, archiving, processing, query, and delivery [62] that has proven its usefulness in supporting data-driven research including big data analysis, machine learning, and data mining [63]–[65]. The Sherlock product is an important component of the ATM research infrastructure used by the NASA Ames Research Center and their partners, but there remain challenges that need to be overcome. These challenges have been acknowledged in presentations given by researchers from NASA Ames Research Center in 2018 and 2019 [67], [68]. The major challenge discussed is that, although Sherlock stores archived data from different data sources, it only allows users to query data from one individual source at a time. This is because Sherlock is not a unified database and the datasets stored are heterogeneous in terms of data formats, spatial and temporal alignment, and scientific units. Because of this, it is hard to bridge across the data from different sources stored in Sherlock during query operations.

Researchers from NASA are not the only ones trying to develop an integrated aviation data warehouse [63], [69]–[71]. Larsen [69] has presented an integrated aviation data warehouse that was developed in support of aviation big data analysis research. The author points out that since aviation data comes from diverse data sources, they do not have the standardization, uniformity or defect controls required for reliable integration. In addition, the diversity of data sources makes the

size of data extremely large for a given period of time, and that might exceed the capability of a traditional desktop to manage and make use of the data.
Tyagi and Nanda [70] presented an architecture for the development of a data warehousing and big data analytics tool for ATM researchers. The proposed tool is designed to be an intelligent repository for a variety of ATM data and would allow users to combine datasets before querying and analyzing them instead of manually downloading, cleaning, ingesting, and querying a small subset of one or multiple datasets. The authors point out that having such a data warehouse can save redundant steps by providing one platform to solve the needs of a larger number of researchers while ensuring accurate, stable, and easily accessible data solutions.

Ayhan et al. [71] describe a novel analytics system that processes, correlates, and stores Aircraft Situation Display to Industry (ASDI) data in a data warehouse. The authors clarify the need for developing scalable data warehouses to better manage and store data. They point out that it is hard to perform analytics on raw data as the collected data is large, compressed, and requires correlation with other flight data before it can be used for analysis. The research group designed and developed a data warehouse to store two years of archived ASDI data. The data warehouse was then used to support the development of software models to predict airspace density as a means of providing more refined rerouting decisions.

In 2020, a group of researchers from Embry-Riddle Aeronautical University (ERAU) presented a research initiative to address not only the problem of managing the accumulated FAA air traffic management data they collect, but also to make the data useful in support of data-driven research conducted at the university [63]. The research group point out that it remains difficult to utilize the collected data because it still requires a series of processes before it can be used. These processes include extracting appropriate data files from the archive, decompressing those files, extracting the relevant data from the decompressed files, and correlating data between multiple files to fill in potentially missing information. The authors emphasize that such a workflow is not suitable for a sustainable research program and is time and resource intensive. The research group also mention that the challenge of making use of available aviation data for aviation big data research is not unique to ERAU, and that the challenge is primarily caused by the wide variety of aviation data types. They believe that there is a need to develop a system-level design for an aviation big data system that can provide adequate and useful support to research and operational needs.

To demonstrate a proof-of-concept, the ERAU research group takes a 24-hour period of collected aviation data from various data sources and develops a data repository to store these data. They

present their workflow and discuss the problems they encounter during the implementation including extracting data from different data sources and converting them to a format suitable for storage; identifying the correlation between different datasets to fill in the gap where there is missing information from one data set; and designing an interface that allows users to query and visualize the data they are interested in. After implementing the proof-of-concept, the researchers present several lessons learned from their research. They emphasize that data clean-up and pre-processing are important after extracting data from data sources, and that it should be expected to see inconsistency in data format across data sources as well as incomplete information within a single source. Finally, they point out that the data repository should be capable of maintaining efficient performance in terms of feeding back the query results to users.

The examples of Sherlock and other data repositories described above reflect their usefulness in terms of supporting aviation data-driven research. The Sherlock data repository developed by NASA aims to provide researchers with a platform to download all ATM-related data [62]. Although Sherlock is considered an essential tool for ATM researchers, it is not a unified data repository and its data cannot be used directly in data-driven research [63], [70] without cleaning and standardization processes being applied [67], [68]. Unlike the Sherlock data repository, there are other researchers who design and develop small data repositories for specific research applications [63], [69]–[71]. These authors prefer to have a data repository with only a limited amount of data defined by the research purpose rather than a master data repository like Sherlock. For this smaller type of data warehouse, raw data extracted from data sources is processed to satisfy the research requirement before storing them in the repository. Characteristics of this kind of data repository are manageability using a normal desktop computer, and the reliability of providing efficient performance when users query data from it.

The literature review reveals that, although we are in the era of big data and that much of that data has relevance for promising areas of aviation research, challenges remain to its utilization. The biggest challenges are related to correlating large amounts of data from a variety of sources, and locating, extracting, processing, and storing relevant material in a format ensuring consistency, compatibility, and ease of use for the aerospace researcher. Although the large amounts of information available have led to the temptation to create large “data warehouses”, the practice of storing large amounts of data increases the difficulties associated with the querying, extraction, and processing tasks necessary to make the data useful to the researcher in an efficient manner.

## 2.7 Research objective

In the study that is the subject of this thesis, a smaller, more efficient, aerospace research-specific data repository is proposed, and a methodology is presented for its development. The methodology is aimed at developing best practices and alternatives for pre-processing and storing data according to a well-defined research need in such a way that the information is accessible, non-redundant, clean, and compatible. The approach is validated with a proof of concept where a prototype data repository is created in support of a defined research project. The research objective can be broken down into five main tasks as follows:

1. Identify aviation data sources that satisfy the given research need within the constraints that the sources must be *publicly available* and *easily accessible*.
2. Design and develop an independent data repository that i) can be handled by a common desktop computer; ii) is compatible (can be accessed) with various programming languages; and iii) can provide effective feedback to user queries.
3. Develop a script to implement Data ETL and extract data from the sources identified in Task #1, clean it to satisfy specified requirements, and load it to the data repository developed in Task #2.
4. Design and develop a Graphical User Interface (GUI) that provides a platform for the data repository user to visualize and interact with the data.
5. Verify the methodology using a proof-of-concept test case.

# 3. Research Methodology

This chapter presents the methodology used to create an aerospace research-specific data repository that satisfies the five main tasks discussed in Section 2.7 and illustrated in Figure 2. Section 3.1 discusses the data source selection based on the defined requirement of a specific research project. Section 3.2 presents the design and development of a data repository that satisfies the requirements of Task #2 as defined in Section 2.7. Section 3.3 presents the implementation of Data ETL as described in Section 2.3.3; and Section 3.4 describes the development of a web-based GUI to provide data visualization to users. Section 3.5 concludes the chapter by presenting a validation approach using a proof-of-concept test case.

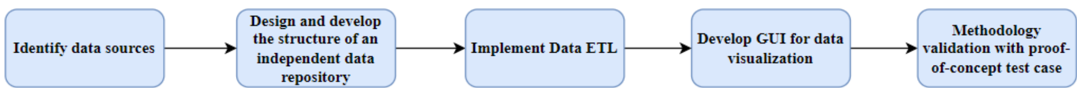


*Figure 2. Workflow*

## 3.1. Data source selection

Task #1 involves the identification of suitable aviation data sources that fit the given research criteria with the added constraint that the data be publicly available at no charge. In the literature review conducted by Li and Ryerson [57], they identify 16 data categories of aviation-related data used in data-driven aviation research, the most commonly used being Air Traffic Control data; individual flight-level data; economics, logistics and operational data;  airport and airline specific data; fuel and fuel-related data; weather-related meteorological data; aviation geography and geometry data; and socioeconomic, demographic and population data. In the study that is the subject of this thesis, four of these categories of data are accessed; individual flight-level data, airport and airline specific data, weather-related meteorological data, and aviation geography and geometry data.  The data are used to implement the research methodology in the context of a specific research project. The remaining part of this section provides more detailed information about the data sources selected and the types of data associated with each of them.

### 3.1.1 Weather data

The NOAA is a scientific agency within the United States Department of Commerce. The role of the NOAA is to monitor natural environmental activities in the United States. One of the organization's responsibilities is to release weather forecasts as well as warnings when weather hazards are predicted [72]. The NOAA collaborates with other top-level organizations to jointly develop weather products for use by the public and federal agencies. These organizations include NASA, FAA, and the United States Department of Defense (DOD) as shown in Figure 3.

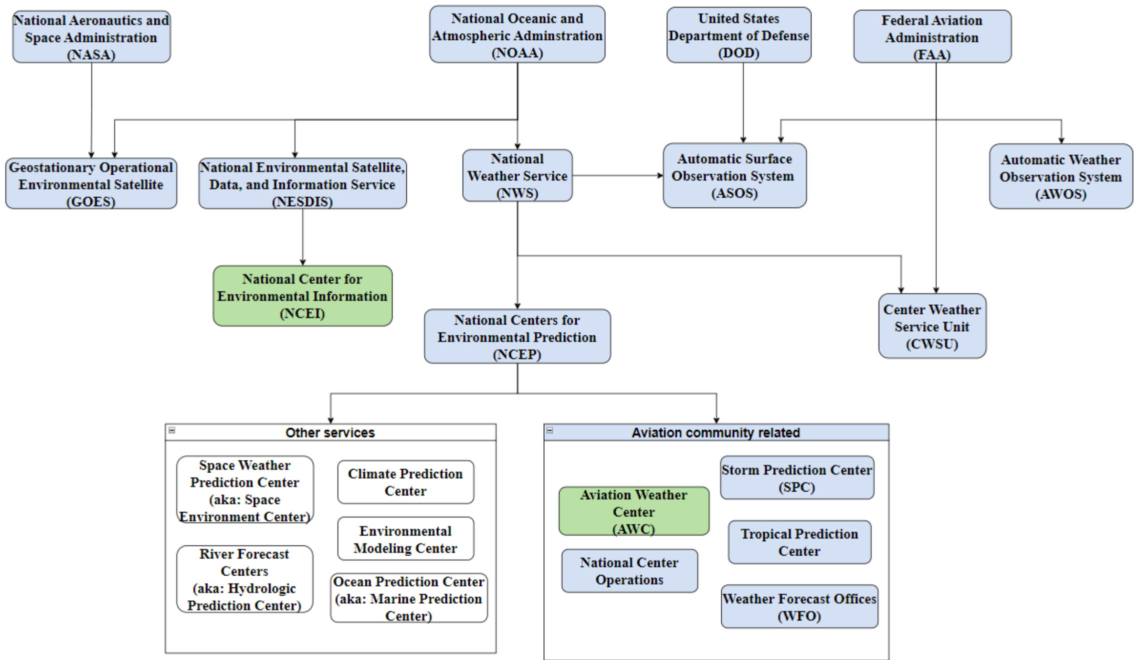


*Figure 3. The service available in NOAA*

There are multiple environmental services associated with the NOAA program, and some of these are responsible for providing information to the aviation community [73]. Figure 3 shows an overview of the services available from the NOAA, where the blue boxes represent services related to the aviation community and the white boxes show other services. The National Weather Service (NWS) is one of the government institutions under the NOAA program, and an important role of the NWS is to issue severe weather warnings to save lives and minimize property loss [74]. Within that context, several weather prediction services under the NWS provide direct weather forecast information to aviation users.

The two green boxes highlighted in Figure 3 are the data sources selected for use in the project that is the subject of this thesis, and both are owned by the NOAA. The Aviation Weather Center (AWC) is an institute belonging to the National Centers for Environmental Prediction (NCEP) with the objective of providing consistent, timely, and accurate weather information to the global aviation community. The National Center for Environmental Information (NCEI) is an organization belonging to the National Environmental Satellite, Data and Information Service (NESDIS) and is the Nation's leading authority for environmental data. The NCEI manages a large amount of archived atmospheric and oceanic data and fulfils the role of helping the NOAA meet the growing need for high value data in environmental research. Unlike the AWC, the NCEI provides only the raw data to users and for data-driven research, raw data is more useful because machines can process complex data values more effectively than humans.

All weather data captured by any institution under the NOAA program is publicly available on the NOAA website. The various types of weather data are collected by different devices, and the raw data are 'encoded' in their own language and appear in different data formats [73] in the different databases. The NOAA's Weather and Climate Toolkit (WCT) is an independent and free software released by the NOAA and has two primary functions: data visualization and data export [75]. The software can generate satellite images from any given raw radar data file but can also convert a raw radar data file to a variety of common formats and export. Researchers can use the WCT software to convert raw weather data files to the desired format compatible with their research activity.

In 2000, NASA conducted a study to review all the aviation weather products available at the time [73]. The study uses Federal Aviation Regulations Part 91 (General Operating and Flight Rules), Part 135 (Air Taxi Operators and Commercial Operators), and Part 121 (Domestic Commercial Operators) as the context to analyze the critical weather information required by aviation users at each phase of flight. Depending on the phase of flight, different weather products may be required, because each product has a different coverage area. For example, while certain weather products are only available in the terminal area, others are issued for all 48 contiguous states in the U.S.. The study identifies weather products used by the aviation community as well as the data sources, data content, and updated frequency of each weather product. Table 2 provides an overview of the weather products studied and is presented here to demonstrate the diversity of data formats, update frequencies, and coverage areas for the different organizations under the NWS.

As an example, the Meteorological Aerodrome Reports (METAR) is a joint effort by the NWS, the DOD, and the FAA, and provides an hourly report of surface weather information for US airport terminal areas [73], [75]. The METAR data is a text string that combines information from Automated Weather Observing Systems (AWOS) and Automated Surface Observation Systems (ASOS) to provide the report issue time, wind speed and direction, visibility, temperature, and other weather phenomena.

*Table 2.Information about different weather data [73]*

| Weather Product | Description | Responsible organization | Coverage Area | Update rate | Format |
|---|---|---|---|---|---|
| Aviation Routine Weather Report (METAR) | Surface condition at the airport | ASOS[1] AWOS[2] HO[3] | Terminal | 1 hour | Text string |
| Terminal Area Forecast (TAF) | Airport terminal weather forecast | NWS WFO[4] | Terminal | 4 hours | Text string |
| Airman's Meteorological Advisory (AIRMET) | 3 categories, hazardous atmospheric conditions for VFR | AWC[5] | 3000 square miles | 6 hours | Text string |
| Significant Meteorological Information (SIGMET) | Hazardous conditions for all user categories | AWC | 3000 square miles | 6 hours | Text string |
| Low-Level Significant Weather Chart (LLSWC) | Aid VFR briefing | NWS | U.S. region | 4 times per day | Graphical chart |

[1] ASOS: Automatic Surface Observation System

[2] AWOS: Automatic Weather Observation System

[3] HO: Human Observation

[4] WFO: Weather Forecast Office

[5] AWC: Aviation Weather Center

| | | | | | |
|---|---|---|---|---|---|
| High-Level Significant Weather Chart (HLSWC) | Provide forecasts during en-route phase for international flight | NWS | U.S. region | 4 times per day | Graphical chart |
| Winds and Temperature Aloft (WA and TA) | Information at 9 discrete elevations from 3000 ft to 39000 ft | NCEP[6] | U.S. region | 12 hours | Graphical chart or text string |
| Meteorological Impact Statement (MIS) | Unscheduled weather information help flight planning, flow control | CWSU[7] | Regional | As condition warrant | Graphical chart or text string |
| Center Weather Advisory(CWA) | Nowcast information help flight crew avoid hazardous condition | CWSU | Regional | As condition warrant | Graphical chart or text string |
| Pilot Report(PIREPS) | Atmosphere observation by pilot or aircraft instrument | Pilots | Localized | As condition warrant | Text report |
| Satellite Imagery (SI) | Images of cloud and the temperature of the cloud | GOES[8] NOAA | National | 15 min ~ 1 hours | Satellite image |
| Radiosonde Additional Data (RAD) | Information on freezing level and relative humidity | NWS | National | 12 hours | Radar image |
| Next Generation Weather Surveillance Radar(NEXRAD) | Produce 18 products related to precipitation and velocity estimates | NWS | ~ 200 mile radius | 6~12 minutes | Binary file |
| Terminal Doppler Weather Radar (TDWR) | Provide wind shear precipitation in the terminal area | FAA NWS | Terminal | As condition warrant | Radar image |

[6] NCEP: National Centers for Environmental Prediction

[7] CWSU: Centers Weather Service Unit

[8] GOES: Geostationary Operational Environmental Satellite

### 3.1.2 Air traffic data

Airline Situation Display to Industry (ASDI) is a data stream that broadcasts real-time air traffic data to members of the aviation community and has been serving as the data feed of FAA's Cooperative Research Data Agreement since 1998 [71]. ASDI delivers information to users through a text string, where the ASDI messages include, but are not limited to, flight plan information, arrival information, and departure information for aircraft in the NAS. Table 3 lists the information content associated with each type of message delivered by ASDI.

*Table 3. Information available in ASDI data stream*

| Message type | Content |
| --- | --- |
| Flight plan | Aircraft identification |
| | Departure point |
| | Destination |
| | Aircraft type |
| | Speed |
| | Coordination Fix |
| | Coordination time |
| | Assigned altitude |
| | Requested altitude |
| | Route |

| | |
|---|---|
| Arrival information | Aircraft identification |
| | Departure point |
| | Destination |
| | Arrival time |
| Departure information | Aircraft identification |
| | Aircraft type |
| | Departure point |
| | Actual departure time |
| | Destination |
| | Estimated time of arrival (ETA) |

The ASDI is a data stream, and the organization is not responsible for collecting the data, but rather makes available data coming from the Enhanced Traffic Management System (ETMS) [76]. The ETMS in turn derives its air traffic information from several sources [77] including airline schedule data from the Official Airline Guide (OAG); real-time NAS messages from the Air Route Traffic Control Centers (ARTCCs); and air traffic data over the contiguous and the oceanic area of the United States from the Aeronautical Radio Incorporated (ARINC) and the Dynamic Oceanic Tracking System(DOTS). The ETMS combines the available data to always maintain a comprehensive picture of air traffic in the NAS. This information is then broadcast in real time by the ASDI and can be publicly accessed by members of the aviation research community.

Another public source of air traffic data is the Automatic Dependent Surveillance-Broadcast (ADS-B). ADS-B uses satellite navigation and other sensors to determine the position of an aircraft and

broadcast it periodically. The latest version of the Code of Federal Regulations Title 14 Part 91.225 and Part 91.227 states that all aircraft must be equipped with ADS-B Out to fly in most controlled airspace after January 1, 2020 [78], [79]. In general, ADS-B Out refers to an avionics subsystem that broadcasts flight information of the equipped aircraft. Any other airspace users equipped with ADS-B In systems can receive the broadcast information. Ground receivers like the Air Traffic Control System can also receive the broadcast information by equipping an antenna with receivers and an adapted surveillance processor. ADS-B is a surveillance service used to support separation assurance and traffic flow management [80]. Unlike the weather products discussed in the previous section that have minute-based or hour-based update rates, the ADS-B message updates every second. The flight information broadcast by ADS-B Out includes time, horizontal and vertical position, speed, barometric altitude, and aircraft identification code.

Accessing the ASDI data stream is more complicated than accessing ADS-B data. The FAA maintains a list of acknowledged direct ASDI subscribers who can access the ASDI data stream [81]. ADS-B data, however, can be accessed through the internet and there are several platforms that provide archived ADS-B data to the public [82]–[84]. All ADS-B data platforms identified in this research are notable for having missing information or aircraft positional error in the data they provide. For the purpose of this research, when these errors are encountered, the records that lack information on aircraft position are neglected during the data extraction process. Section 4.3.1.2 present more detail on the work associated with obtaining and processing ADS-B data.

This section has provided a brief presentation of some of the wide variety of available data sources on weather and air traffic data. The selection of the type and source of data used on any given research project will be driven by the requirements of the project itself and limited by the constraint that the data be publicly availability and easily accessible.

## 3.2 Independent data repository

Task #2 from Section 2.7 concerns the design and development of an independent data repository that i) can be handled by a common desktop computer; ii) is compatible (can be accessed) with various programming languages; and iii) can provide effective feedback to user queries.

While this task is defined independently, it is performed iteratively with Task #3 – the implementation of data ETL. Once the data source has been selected (Task #1) and the data

downloaded and decoded (Task #3), it must be cleaned to meet the research project specific requirements (Task #3) and then uploaded to the new database (Task #3). The processes used for cleaning the data will depend on the structure and organization of the raw data, and the type and structure of the database required to store the cleaned data (Task #2) will depend on the approach to data ETL.

The data content of the weather and air traffic data sources discussed in Section 3.1 can be considered structured data, where structured data refers to any data converted to a predefined structure and format before being placed in a storage location [85]. Structured data is often described as data, especially numbers or text strings, that can be organized into tables or spreadsheets, whereas unstructured data is information that cannot be arranged in such a way including images, audio and video files. Figure 4 offers more detail on the differences between structured and unstructured data.

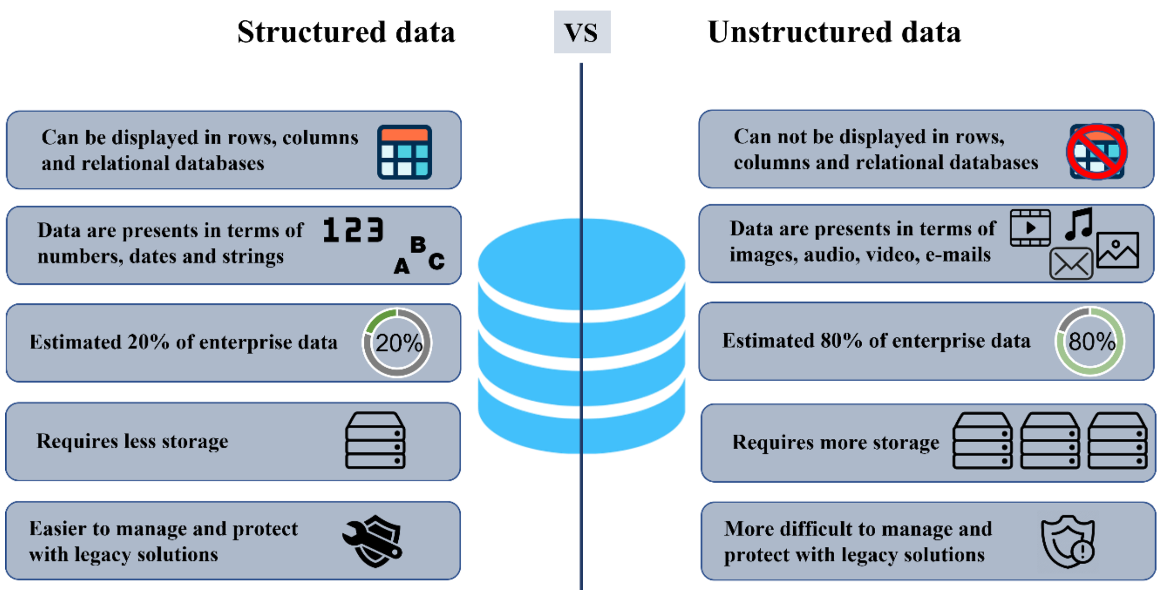


*Figure 4. Structured data vs unstructured data (Adapted from 'Structured Data vs. Unstructured Data: what are they and why care?', Lawtomated [85])*

A selection of different types of databases and data management systems was presented in Section 2.4 with respect to their advantages and disadvantages. A common choice for structured data is the

relational database. Relational databases maintain data in tables, providing an efficient and flexible way to store and access structured data. When designing a relational database, the primary goal is to minimize duplicate data columns among different tables and build connections between them by identifying their correlation. This requires an understanding of the data content as well as the requirements specific to the defined research project. Based on the type of data obtained from the data ETL (Task #3), the process begins with questions that may include “What does the cleaned data look like?”, “How can the data obtained be structured in separate tables?” What criteria should be used to create relationships between tables?”, etc.

The structure of the database used for the proof of concept that is the subject of this thesis is described in detail in Section 4, where the answers to the questions posed above are provided for a selected case study.

## 3.3 Data ETL

The third task identified in Section 2.7 is the application of data ETL techniques to extract data from the sources identified in Task #1; transform it to satisfy specified research requirements; and load it to the data repository developed in Task #2. If the raw data is ‘encoded’, it must be decoded, and the underlying structure and organization examined. The decoding process is driven by questions such as “How are we going to use the data?”; “Is the original structure adequate?”; and “How do I want the data stored in the new database so that it is easy to use?”.

### 3.3.1 Data Extraction

Different data download methods are needed to assist the data extraction process depending on which data sources are selected. This section introduces three data extraction methods that were investigated in this research. Section 3.3.1.1 introduces the most common data extraction method, and Sections 3.3.1.2 and 3.3.1.3 present high-level descriptions of two data extraction methods implemented for the case study that is part of this thesis and are described in more detail in Chapter 4.

### 3.3.1.1 Direct download

The most common way to download something from the internet is by direct download. This data extraction method can be applied to data sources that provide users with a GUI. Users are capable of viewing or selecting the data of their interest through the GUI and can choose the file to download from there.

Figure 5 shows an example of downloading Next Generation Radar (NEXRAD) data from the NCEI website. The NEXRAD data collected at the KJFK weather station on January 1st, 2020 from 00:00:00 to 00:59:59 are the subject of the example. The data file identified in the search can be downloaded from the website by clicking the 'Download' button located at the bottom right corner. The NOAA has several direct download platforms where users can obtain data by selecting a country, city, zip code, or any geographical point on a map [86].

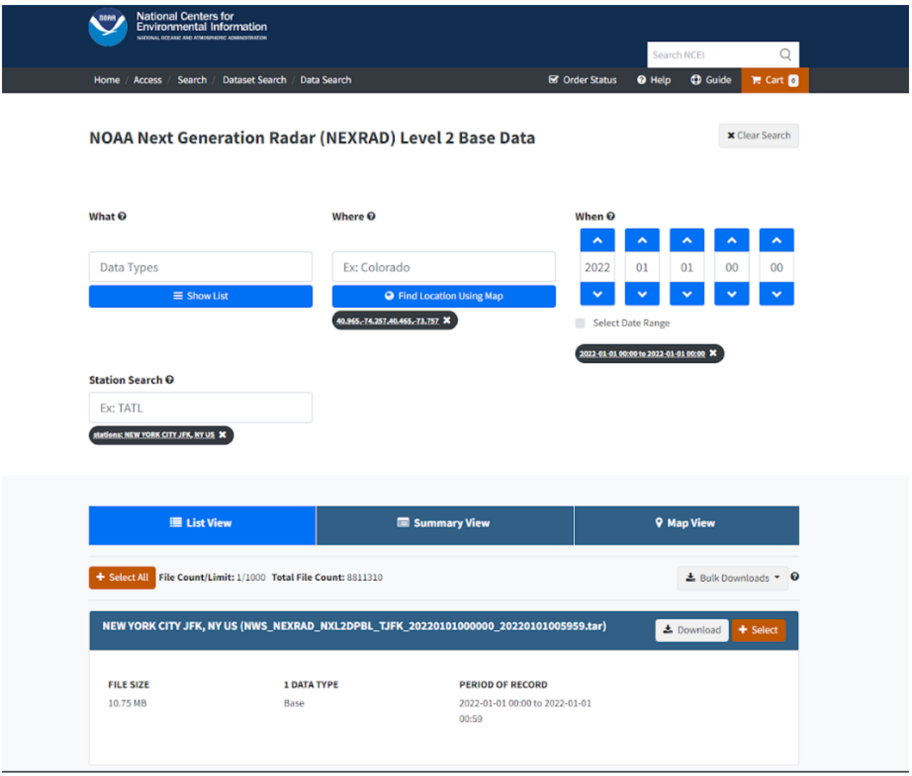


*Figure 5. Direct download NEXRAD data from NCEI[86]*

Direct data download has the advantage of being user-friendly, particularly because the GUI provides guidance for searching and finding the specific dataset the user requires. The disadvantage of this method is that it is difficult to automate the data extraction process and download a wide array of data files at the same time because it requires the user to click and download one file at a time. This disadvantage could be overcome by using application programming interfaces (APIs) or port connection if the selected data sources have these services available. The following sections will introduced two data extraction methods that uses APIs and port connection.

#### 3.3.1.2 Amazon Web Service Command Line Interface

Amazon Web Services (AWS) is a subsidiary of Amazon that provides businesses, governments, and individuals with on-demand cloud computing platforms and APIs. AWS offers a storage service named Amazon Simple Storage Service (Amazon S3). Amazon S3 uses an object storage architecture which treats data as an object and can organize many objects amongst different "buckets" [87].

Figure 6 illustrates the overall architecture of the Amazon S3. Each data object includes the data value, an object key that works as a globally unique identifier, and a metadata capability for storing extra information. A bucket container is the storage entity for storing data objects, and multiple data objects can be stored in the same bucket container. When a bucket container is created, the bucket owner is required to assign a name to the bucket and choose an AWS region. The AWS region is a geographical location where the AWS cluster data centers [88]. The AWS region is an important piece of information for identifying buckets. The bucket owner can restrict access to a specific group of users or open the bucket to the public.

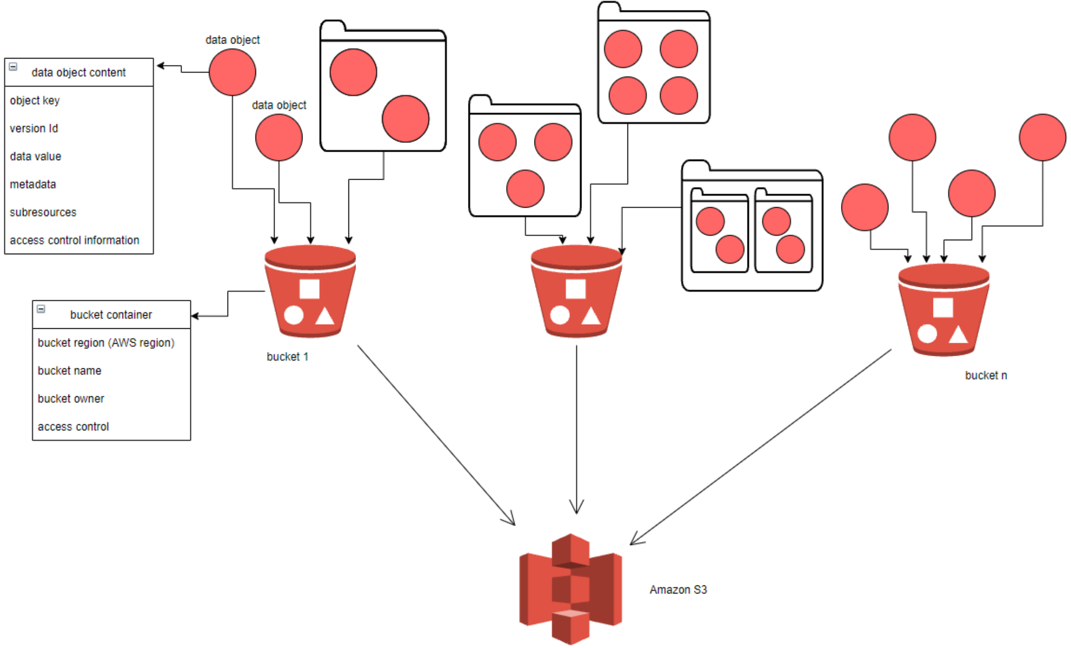


*Figure 6. Amazon S3 structure*

The AWS Command Line Interface (CLI) is an open-source, unified tool for AWS service management. The data object stored in Amazon S3 buckets can be accessed using AWS CLI to connect and send commands to the AWS server. The AWS CLI can be downloaded from AWS's official website and is compatible with the three most popular operating systems: Microsoft Windows, macOS, and Linux [89]. In order to use the AWS CLI to access data stored in Amazon S3 buckets, users are required to specify the AWS region of the bucket as well as the data output format. Figure 7 shows an example of AWS CLI initial configuration, where the two access keys shown in the figure are used to verify whether a user has the authorization to access restricted buckets. To access buckets that have open access, there is no need to specify the two access keys in the initial configuration.

```
$ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: json
```

*Figure 7. AWS CLI initial configuration*

Table 4 shows the command syntax used in AWS CLI to access and download data objects from Amazon S3 buckets. In the command syntax, <bucket> represents the name of the bucket, <prefix> represents the name of folders stored inside the bucket, <object> refers to the globally unique identifier of the data object, and users can define the destination of the downloaded data by configuring <target>.

*Table 4. AWS CLI command syntax*

| **Command syntax** | **Description** |
|---|---|
| *aws s3 ls <bucket>* | List all objects and prefixes in <bucket> |
| *aws s3 cp <bucket>/<object> <target>* | Copy a <object> from <bucket> to destination <target> |
| *aws s3 cp <bucket>/<prefix> <target> --recursive* | Copy ALL <object> in <prefix> from <bucket> to destination <target> |

Figure 8 shows an overview of the complete process of using AWS CLI to access and download data objects from Amazon S3. The advantage of this data extraction method is the ease with which the process can be automated to download a large amount of data. However, this can only be achieved under two conditions. The first is that the user must know the basic information about the bucket they want to access including, but not limited to, the name of the bucket; the AWS region of the bucket; and how the data object is organized inside the bucket. The second condition is to obtain access permission to the bucket. Depending on how the bucket owner configures the access control of their bucket, some buckets have restricted access while others are publicly accessible. Some companies and organizations configure their buckets with restricted access so that the data object is only available to their members.

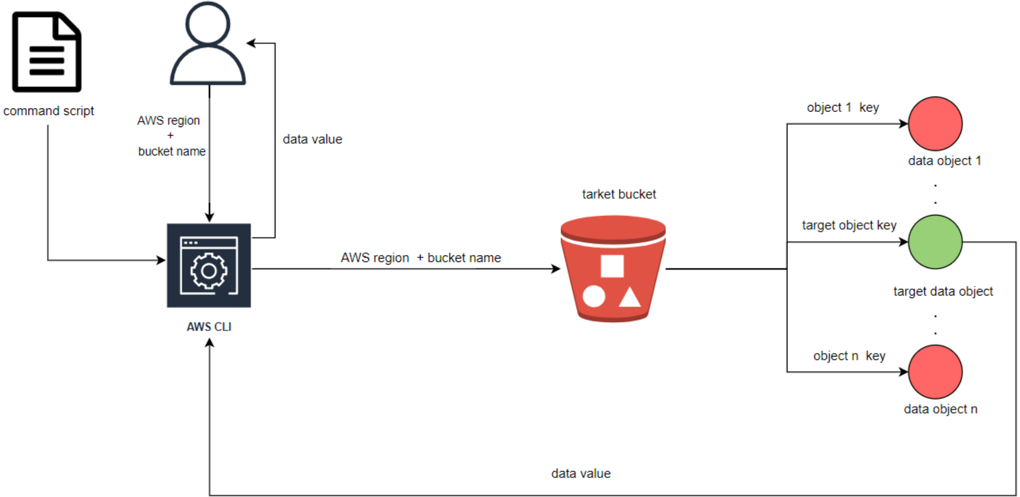


*Figure 8. Accessing data from Amazon S3 using AWS CLI*

### 3.3.1.3 Apache Impala

Apache Hadoop is a software platform that manages data processing and storage for big data applications. It is credited with being the platform for modern cloud data storage because it breaks large data sets into manageable pieces and distributes the data analysis work to different computing clusters. The data stored in Apache Hadoop can be queried and accessed by using an open-source SQL query engine named Apache Impala [90].

The Impala Shell is a tool that can be used to communicate with Apache Impala by using text-form commands. The Impala Shell connects to the Impala service through Secure Shell (SSH), a network protocol that allows one computer to remotely access another computer or server over the internet. The SSH is known for its identity authentication and encoded data communication, ensuring security in the communication between the end-user and the database server. To use Impala Shell to establish a remote connection to the database server, the user must provide information on the server address, server port, and user identity. In Python, there exist open-source libraries that allow users to establish a connection to a remote server and perform impala-shell operations [91].

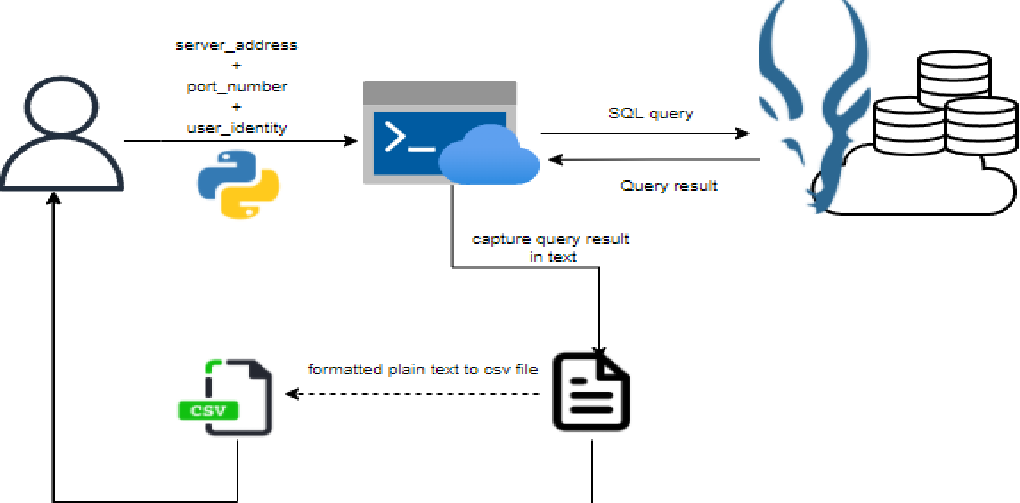


*Figure 9. Accessing data using Impala Shell*

Figure 9 illustrates the process of using Impala Shell to access data on a cloud data server. Users can access an online database server, and once the client is connected, the client can use SQL queries to search and explore the database. SQL is a query language used for managing relational databases. Using SQL queries allows users to extract data from a remote data server and once the query execution is done, the query result will be returned to users in plain text. To make the data transformation process easier, the query result is converted from plain text to comma-separated values (CSV) files without modifying any of the data content. One way this can be achieved is by using a Python library named Pandas designed for data manipulation and analysis. Additional discussion on how to use SQL to query data from a relational database can be found in Section 3.4.

The advantage of the Impala Shell is like that of the AWS CLI, in that it facilitates automated data extraction. Users can create a script with a list of SQL query statements and pass the script document to Impala Shell to automate the data extraction process. A disadvantage is that this method requires a stable internet connection. If a download process is interrupted by a loss of internet connection, it is impossible to locate the breakpoint and restart the download process at the breakpoint. In addition, the method also requires the user to know the address of the remote data server and the server port open for connection.

#### 3.3.1.4 Summary

This section introduced three methods that can be used to extract data from data sources depending on how it is presented to users. This is the first step of the data ETL defined in Task #3 in Section 2.7. The following section will introduce the second ETL step – the process of transforming the extracted raw data into a form that satisfies the given research requirements.

### 3.3.2 Data transformation

After extracting data from a data source, the next step is data transformation. In data ETL, data transformation is the process of converting data from one format or structure to another [41]–[44]. Raw data is not always readable or understandable to humans because the format depends on the way it was collected, particularly if it was collected by machine. If that is the case, the first step in data transformation is to convert the raw data into a readable form, and data owners often provide information about how to do the conversion on their website. For example, the NOAA's WCT mentioned in Section 3.1.1 is a software that can convert raw weather data files to various common data formats.

The WCT can generate radar images from a raw weather radar file, as well as decoding and exporting the input radar data to a specific file format. Figure 10 is an example of using the WCT GUI to generate the radar image and to export the decoded data to a CSV file with a given raw weather radar file. The toolkit can be executed either through a GUI or through command lines. The user can choose the execution method depending on whether the purpose is to observe one specific file or to decode and export the data of a large number of raw weather radar files.

The decoded data is first converted into CSV files so that the researcher will be able to read and explore the raw data content. The CSV file format is selected as it allows data to be saved in a tabular format, which is similar to the tables within a relational database. The advantage of converting the decoded raw data to CSV file format is that it makes it easier to import the data file to another storage database in the next step.

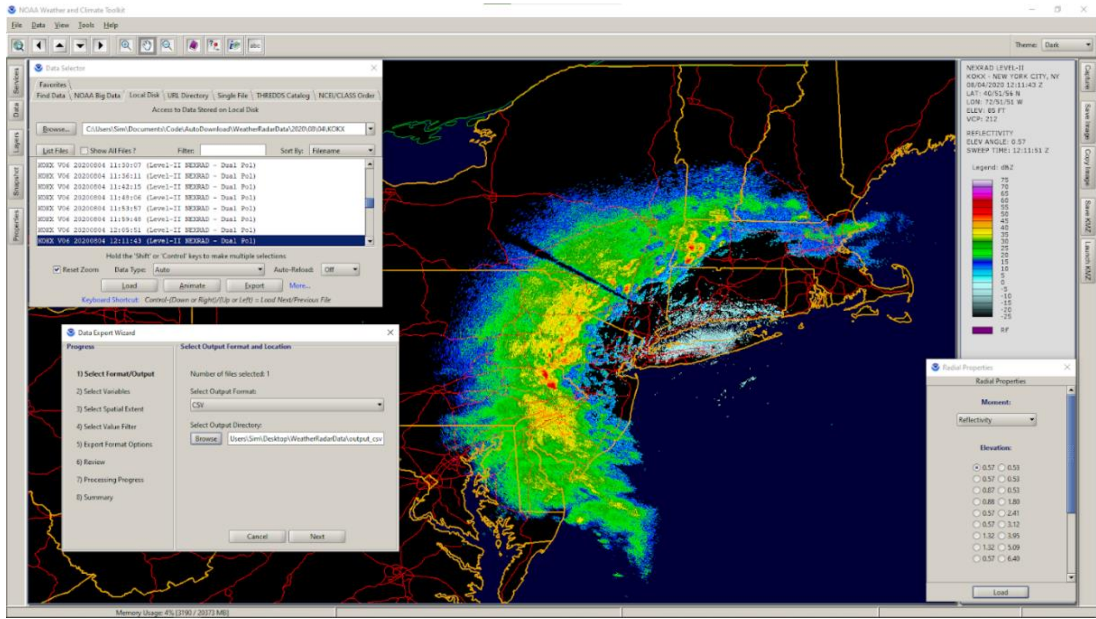

*Figure 10. Decoding raw radar data file with WCT*

Once the raw data set is converted to a CSV file, it is ready for the necessary data transformation operations. Data transformation involves the process of cleansing, filtering, reorganizing, and converting raw information into a desired format [41]–[44]. Figure 11 illustrates the overall process of data transformation.

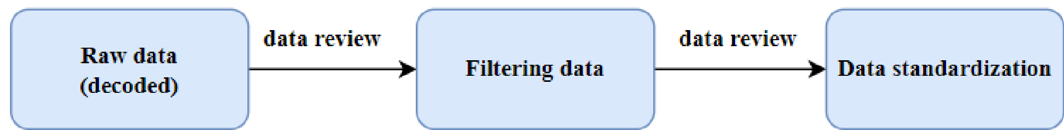


*Figure 11. Data transformation process*

Once the raw data has been converted into a readable format, the data content can be explored. The raw data extracted may contain more information than the research requires, and redundant information is a waste of storage capacity in the project data repository. If that is the case, the raw data content is filtered to retain only the required information.

If more than one data source is used, the data transformation phase will also require actions to standardize the common data content after filtering out undesired content. In this research, data

standardization refers to the process of identifying duplicate data contents in different datasets and unifying their naming convention and data unit. Figure 12 shows an example of the complete data transformation process with two raw datasets.

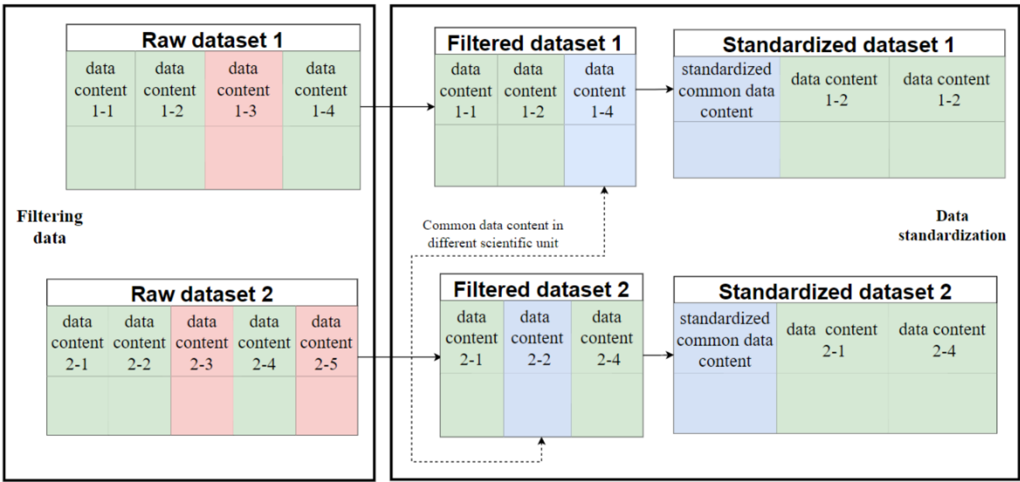


*Figure 12. Data standardization with two raw datasets*

In the example shown in Figure 12, the red column in the raw datasets represents the undesired data component that is not needed for the given research requirements, and the blue data column in the filtered datasets represents a duplicate or redundant data component. The data standardization process is done for each raw dataset separately but based on the same research requirements since the two raw datasets contain different data contents. The end product obtained after this phase is two standardized datasets that include only the data content that is required for the given research project with no duplication.

A specific example of data that requires transformation because of common content is the time the data is collected. Depending on how the raw data is managed by the data owner, this information can be presented in different ways. One representation commonly used for date and time-related information is defined by the International Organization for Standardization (ISO). The international standard to represent time defined by the ISO 8601 is: yyyy-mm-dd Thh:mm:ss∓UTC offset [92]. On the other hand, for computer operating systems, the Unix time (or POSIX time) system is used. The Unix time is the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970, excluding leap seconds [93]. The Unix time system is more

commonly used in computers as it can represent time as an integer which makes it easier to parse and use across different software systems.

The goal of data transformation is to apply a series of actions to the extracted data to eliminate duplicate information, convert it to a format that satisfies the given research requirements and prepare it for loading into the final storage place. Once the transformation process is completed, the cleaned data is ready to be loaded into the selected storage place.

### 3.3.3 Loading and data storage

The last step of the data ETL process is to load the data into the selected storage place. In the case study that will be discussed later in this thesis, the selected storage place is a relational database as discussed in Section 3.2 that is designed and developed as part of Task #2. The last step of the data ETL process is to load the data file to the predefined database.

For example, the Python programming language provides library extensions that can help convert data in a CSV file to a local database file or to upload the data to a database server. Figure 13 below shows an example of the loading process. The left part of the figure represents three standardized data tables from three separate CSV files where the blue column represents a data component that is shared by all the datasets, the green column represents a unique data component within that dataset, and the purple column represents a data component that is shared by some dataset.

The right part of the figure shows a relational database structure design based on the data correlation, which was determined in Task #2. The main table is designed to store two common data contents that are shared by all three standardized datasets and a *'main_id'* is assigned to each row. Because of the uniqueness of the *'main_id',* it can be used to replace the duplicate information in the three original datasets. Therefore, in the database, the blue columns are removed from each table and replaced by a new column to store the *'main_id'* value. In spite of the fact that the tables have been modified, they still contain the same information as they did before moving to the database. The information in the standardized datasets 2 and 3 can be merged into the database table 2 as these two datasets have another shared data content. Having the information merged into one data table will make querying easier as it will only be necessary to search one data table to retrieve information from two different datasets. The loading process in this phase will be loading the data from the CSV tables to a relational database with the help of Python library extensions.

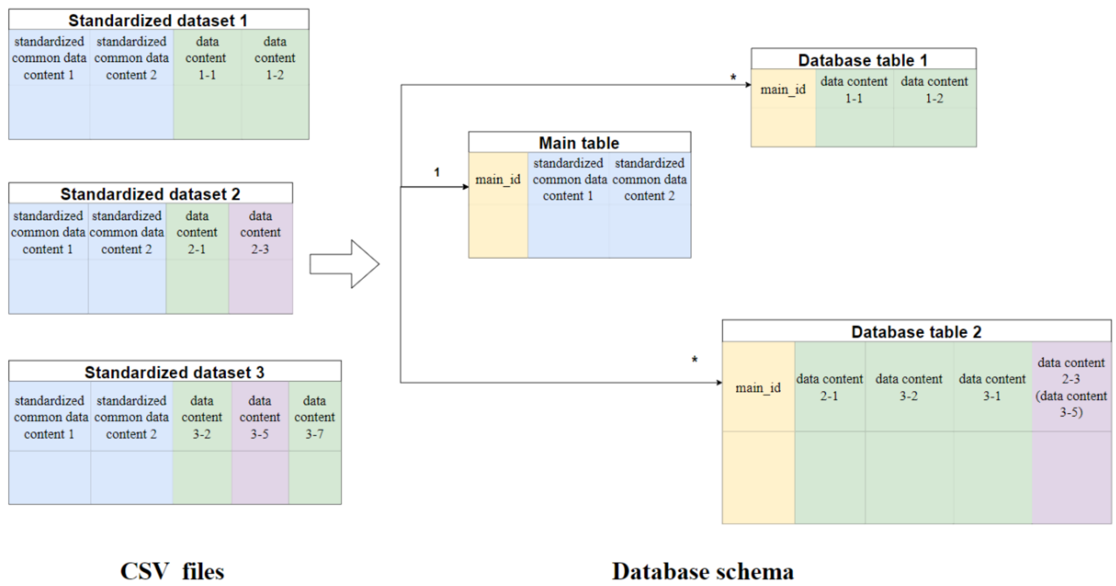


*Figure 13. Load cleaned data to storage*

The process of loading the cleaned data to a final storage place is the last step of the data ETL process defined in Task #3 in Section 2.7. Once the data is loaded to the storage place, it is ready to be used for specific applications defined by researchers. The following section will introduce the design and development of a GUI that allow researchers to visualize and interact with the data for specific applications.

## 3.4 Data visualization and user interaction

Once the data ETL process is completed, the cleaned data are stored in the new database and are ready to be used. The fourth task identified in Section 2.7 is to design and develop a GUI that provides a platform for the database user to visualize and interact with the data. Literature reviews [31], [32] show that data visualization is an effective tool to help humans from a variety of disciplines understand complex information. The selection of the type of visualization model is based on the user's objective [35]. A simple example of data visualization is the Microsoft Excel spreadsheet. Excel can convert data into line charts, bar charts, pie charts, or maps (if the given data is geographically based).

Graphical representations can be in either static or dynamic, where static visualization is a still image that is only able to deliver a message that does not change over time. A dynamic visualization is an animation of a series of data images and often allows users to interact with it. An advantage of dynamic visualization is that it can reveal information contained in the data that is not evident in static displays.

Commercial software such as Microsoft Excel, Tableau, Infogram, and ChartBlocks are data management tools that have been developed for general applications and often cannot satisfy complex and specific research requirements.  Programming languages like C++ and Python have library extensions that can create static or dynamic visualization from a given set of data. Programming languages can be used by researchers to develop GUIs to satisfy specific research requirements. The design and development of the visualization interface depends on the selected data as well as the purpose of the research objective. Depending on the purpose of the research, the visualization tool should be able to address questions like: "What are we using the data for?", "How do we want to display the data?", "What information are we trying to get from the data?", or "What kinds of interaction is needed to help understand the data?", etc.

*Table 5. SQL query statement*

| **Query statement** | **Description** |
|---|---|
| Select | Select rows from one or many database tables and returns this data in the form of a result table |
| Where | Filter database rows based on specific range or value |
| And, Or, Not | Operators that used combine with "Where" |
| Join | Combine rows from two or more tables, based on a related column between them |
| show tables | Show all tables within the database |
| describe table | Show the name of all columns in table |

For the case study that is presented in Chapter 4, the data shown in the visualization tool is the cleaned and standardized data stored in a relational database after the data ETL process of Task #3

described in Section 2.7. SQL is a domain-specific language designed for managing data in a relational database. SQL queries can be used to search and extract data from relational databases for applications. Programming languages like C++ and Python all have library extensions to support the use of SQL queries within the development environment. Therefore, regardless of the choice of programming language for developing the visualization tool, data stored in a relational database can always be accessed and extracted to create graphical representations. For example, Table 5 presents some useful SQL query statements which can be applied for the purpose of exploring the database content or to load data from the database for applications.

Despite existing commercial software that can create graphical representations of datasets, a data visualization tool that can be tailored for specific implementations is developed. The development of the visualization tool is equally important to the database as it can be used to study and analyze data from a specific disciplinary project perspective. The data visualization tool can also serve as an analysis tool to test research hypotheses by testing if the data supports the proposed research approach and/or solution. Once the visualization tool has been implemented, the completed process can be verified using a proof-of-concept case study as described in the following Section.

## 3.5 Process verification using a proof-of-concept test case

This chapter discusses the implementation of Tasks #1 to #4 as defined in Section 2.7. The methodology concludes with Task #5 which is the process of verifying the data selection, cleaning, database design and the visualization tool using a proof-of-concept test case. The purpose of Task #5 is to make sure that i) the database obtained has enough data to support the needs of the defined research; and ii) the data visualization tool developed in Task #4 can serve as a platform that allows researchers to appropriately understand, visualize and interact with the data.

The implementation of the tasks described in this chapter is an approach to designing and implementing a database based on given research requirements. Once the process is complete, a research-specific relational database has been created and includes a data visualization tool that is specifically designed for the project. If the given research requirements change during one or more phases of the research project, the systematic approach that has been developed can adapt to the change in an iterative manner. Figure 14 illustrates how typical changes to project requirements might impact the individual tasks and how the remaining processes would be updated to produce a new database or data visualization tool adapted to the new requirements.

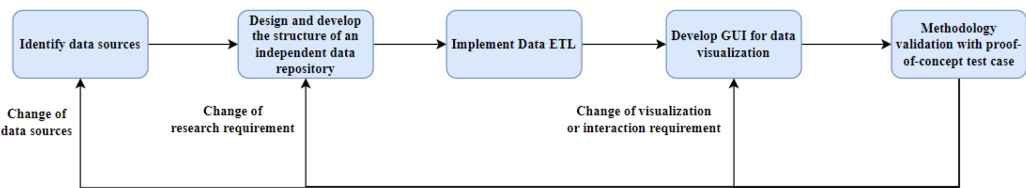


*Figure 14. Effect on tasks due to change of decision from "research requirement" to "data format"*

This chapter presented a systematic methodology for the creation of a database and a data visualization tool tailored to the needs of specific research projects. The approach can be used to structure databases for any data-driven research that requires a series of specific types of data. It is an iterative process that is developed based on given research requirements and that can adapt to changes to requirements that may arise during the research project. A proof-of-concept implementation of the methodology is described in the following Section, in which a database is created to support the requirement for a series of "real world" historic simulated flight scenarios with a range of visualization and data acquisition capabilities.

# 4. Case study implementation

A proof-of-concept implementation is provided in this Section. The case study was performed in collaboration with a co-researcher and is based on their research project investigating risk assessment methodologies for an airspace shared between crewed and un-crewed aircraft. The research required a database that contains ADS-B and weather radar data as a tool for creating a simulated airspace environment on which the risk assessment is based. The researcher provided the types of data required as well as the following criteria for the database:

1. The database must contain enough data to support the creation of a series of varying 'real world' simulated flight scenarios based on historic traffic patterns.
2. The simulated airspace environment must contain the following entities:
    1. Air traffic
    2. Airspace and airport infrastructure
    3. Ground obstacle
    4. Terrain
    5. Weather
3. The database should maintain a high level of efficiency in terms of data acquisition.
4. A data visualization tool is required to visualize and interact with the simulated airspace environment.

In this chapter, Section 4.1 presents the choice of data sources and Section 4.2 describes the design of the database structure. Section 4.3 outlines the process of implementing the data ETL process on the raw data sources identified in Section 4.1. The programming language Python is used in this case study, and the Python code used for the case study is attached in the Appendices. In Section 4.4 the GUI developed for data visualization and interaction is presented and discussed. The final data visualization product developed in this case is a local-host website that is developed using JavaScript and a link to a You-Tube presentation is provided in the Appendices.

## 4.1 Data source selection

The entities required for the simulated airspace environment are defined in Criteria #2, above, and are listed in Table 6, where they are divided into two categories based on their rate of change with respect to time: dynamic and static.

*Table 6. Dynamic entity and Static entity*

| Dynamic entity | Static entity |
| --- | --- |
| Air traffic | Airspace and airport infrastructure |
| Weather | Ground obstacle |
| | Terrain |

Among the five entities in Table 6, airspace and airport infrastructure, ground obstacles, and terrain are classified as static entities. The term 'static' means that the characteristics of these entities change at a frequency that is much lower than that of entities classified as dynamic, such as the position of an air vehicle or a weather system. For example, the FAA releases a dataset every 28 days that is updated to reflect modifications to so-called static entities, for example, an airway or the airspace infrastructure.

In the case-study implementation, only the most recent update of the static entity data is used. The data for the case-study static entities is obtained from a map development tool named Mapbox studio [94] and will be discussed in more detail in Section 4.4. For the purposes of the case-study, the methodology developed in this thesis is applied to create a database storing real-life weather radar and air traffic data collected and archived from two open sources: ADS-B data from the Opensky Network and weather radar data from the NCEI.

The OpenSky Network [82] is a non-profit organization founded in 2012 to provide secure and reliable real-world air traffic data to the public. The organization gathers data using ADS-B transponders which can collect air traffic information every second. The collected data is decoded and archived in a large historical database. The OpenSky Network offers access to its online database free of charge, and users can assess their historical database by establishing an impala-

shell connection. The ADS-B data stored in the database are already decoded, and the data content of the downloaded data will be described in more detail in Section 4.3.1.2. Because the data obtained from the OpenSky Network has already been decoded, there is no need to perform the decoding process in the data transformation phase for the ADS-B data.

The weather radar data used in the case study is the NEXRAD data archived by the NCEI. The NCEI is an organization that helps the NOAA manage archived atmospheric and oceanic data to support the need for high-value data in environmental research. Weather radar data are archived based on the time and the weather station where the data was collected. This makes it possible to extract only the data that are within a specific geographical boundary or collected at a specific time. The NCEI makes its database accessible to the public through multiple platforms, and for the case-study described in this chapter, radar data was extracted from the NCEI Amazon S3 bucket. The advantage of selecting this particular data source is the potential to automate the data extraction process and easily download a large number of data files.

## 4.2 Database structure design

This section discusses the design of the database structure for storing the data required to create the simulated airspace environment. The goal is to have a research-specific database that not only contains sufficient data to support the research but also maintains high efficiency when querying data from the database. The database designed for the purposes of the case-study is based on the data content of weather radar and air traffic data.

The process of developing the database is an iterative process and involves understanding how the data is presented at source while also considering the user's research requirements. A relational database is used for this case study because the selected data are structured. In this case study, the update frequencies of the two selected data sources differ and merging the two datasets will result in missing values in data rows. As a result, ADS-B data and weather radar data will be stored separately. Figure 15 illustrates the design of the database schema. The ADS-B data obtained from the OpenSky Network is already decoded, and the data content of the weather radar data from NOAA is obtained by performing a complete data transformation process as described in Section 4.3.

The design is based on two types of data content common to both datasets: geographical locations and time. Latitude and longitude information is divided into one-degree by one-degree data points

and stored in the 'Geographical_point' table with a given 'Geo_ID' that serves as a unique identifier for each row. Altitude is not included in this table but is managed by creating a new data content named 'geo_BLOB' with the help of an SQLite spatial extension. This approach is discussed in Section 4.3.2.4 as part of the 'load data to storage' phase.

A unique identifier 'Geo_ID' is used to identify the data within a specific geographical region among the two datasets. The purpose of creating the 'Geographical_point' table is to: i) narrow down a range while querying data from the database and ii) extract data that are from the same geographical area across two tables by using SQL queries. However, for the purpose of creating the simulated airspace environment, precise geographical information is required for each entity. Therefore, the latitude and longitude information associated with each data row remains as this information stored in the two tables is in decimal degrees detailed to 6 decimal places.

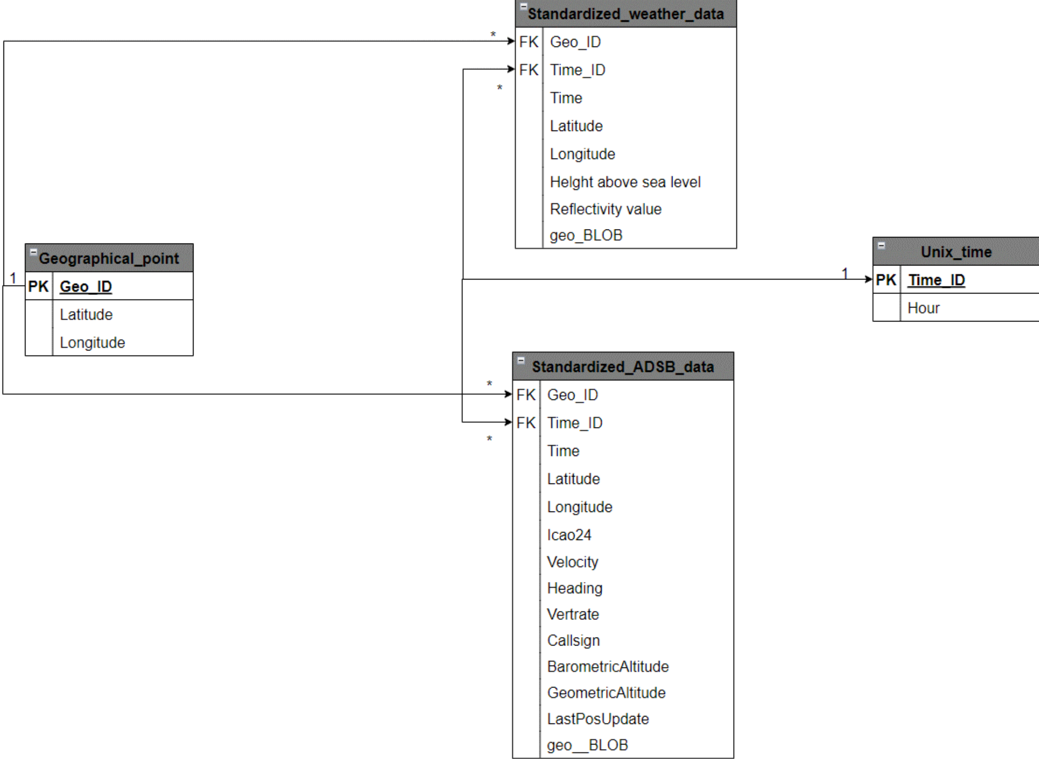


*Figure 15.Database schema*

In a manner similar to that used to create the ‘Geographical_point’ table, a ‘Unix_time’ table is created to assist in querying data collected at specific times in the two standardized tables. The Unix time format is selected in this case because it is the easiest for computers to process. The time difference between each data row in the ‘Unix_time’ table is one hour, and a unique identifier, ‘Time_ID’, is assigned to each data row. In the two standardized tables, each row is assigned the corresponding ‘Time_ID’ based on the hour when data is collected. However, the two standardized tables still have a ‘Time’ content which indicates the exact time when the data is collected.

## 4.3 Data ETL

This section presents the implementation of the data ETL process using the programming language Python. The results obtained from each process is presented at the end of each subsection.

### 4.3.1 Data Extraction

#### 4.3.1.1 Extracting weather data

The weather radar data used in this case study is extracted from the NCEI Amazon S3 bucket. The data is categorized into weather station folders and then stored in subfolders of year, month, and day depending on when the data was collected. Figure 16 illustrates how the archived radar data is stored in the folders of the Amazon S3 bucket.

The AWS region and the name of the bucket can be found on the AWS open data registry website. After obtaining the necessary information about the bucket and how the data is being stored and organized, a Python script was developed to extract weather radar data from the bucket using AWS CLI. The research requirements specific to the case study is for selectable weather “scenarios” featuring extreme weather conditions for specific airspace locations. For this reason, the dates for a selection of significant weather events were obtained from a list provided on the NWS website[95]. These dates were used as a guide for downloading weather radar data collected on different days with varying weather conditions in a way that is useful for the simulated airspace environment.

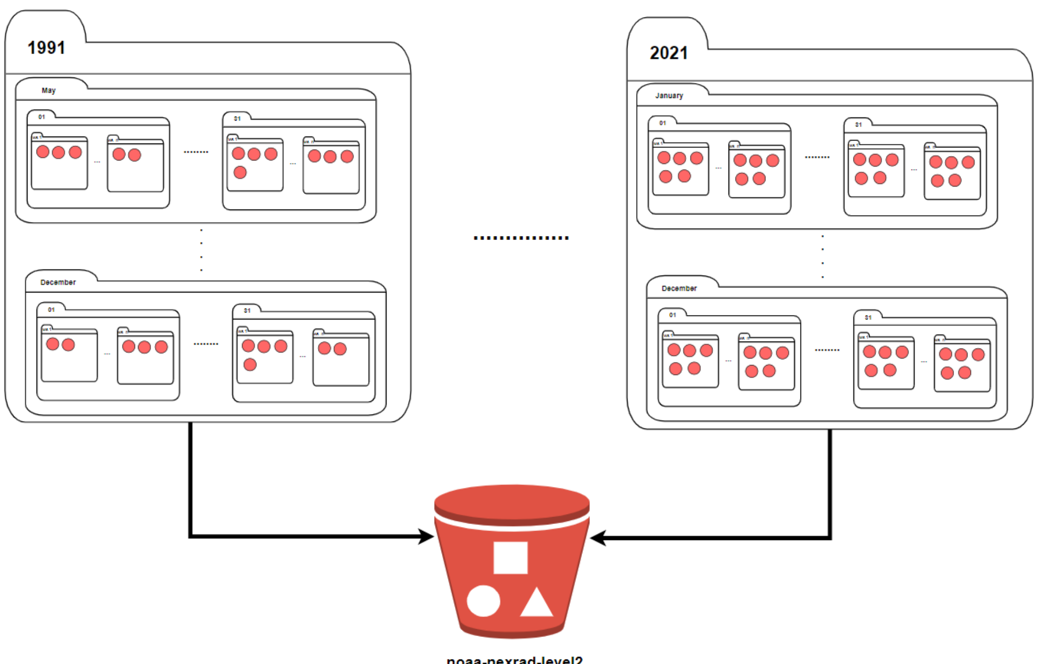


*Figure 16.NCEI Amazon S3 bucket*

The size and number of weather data files obtained from the bucket will differ depending on the location and weather conditions of the selected weather station. This size difference can affect the time required to download the data, and the performance of the computer and the quality of the internet connection can also contribute to the time required. While it is difficult to be specific on how long it takes to download a certain amount of weather data, two examples of data download duration, files available, and the size of all downloaded files are shown in Figure 17 for 24 hours of weather data for two different dates collected by the same weather station (KOKX). Figure 17 indicates that if there was bad weather on the selected date and location, more weather information is available. Despite this, downloading weather data from different days does not significantly increase the download time because the bucket only contains raw files that are compressed. The Python script developed for the data extraction process can be found in Appendix A.

```
Weather data time: 2020/10/03  00:00 - 23:59 UTC
Weather station: KOKX
Data download duration:  0.0  minutes 20.870052  second
Number of files:  175
Size of all files:  490  MB
```
Normal day

```
Weather data time: 2022/04/17  00:00 - 23:59 UTC
Weather station: KOKX
Data download duration:  0.0  minutes 33.459966  second
Number of files:  212
Size of all files:  966  MB
```
Easter tornado event

*Figure 17.Downloading weather data from two different days*

### 4.3.1.2 Extracting ADS-B data

The OpenSky Network was selected as the best data source to satisfy the need for air traffic data as defined in Criteria #2. The historical database of the OpenSky Network is based on Cloudera Impala, and users can connect to the database by establishing an impala-shell connection. The organization provides information on an available open-source Python wrapper named 'pyopensky' that can be used to access and download data. An attempt was made to use this Python wrapper to extract ADS-B data from the OpenSky Network, and the Python script for this implementation can be found in Appendix B1.

Table 7 shows the data content and the description of each term contained in the ADS-B data extracted from the OpenSky Network historical database using the wrapper. The data contains information which is not required for the specified research project where only the data content highlighted in green in Table 7 is required for the research. Another problem associated with using the wrapper to directly download all ADS-B data is that the data obtained may contain aircraft identification or position error. An initial attempt was made to use the wrapper to download all ADS-B data on the selected date and geographical region. The dataset obtained contains redundant information and would require additional processing in the data transformation phase.

*Table 7. ADS-B data content from the OpenSky Network*

| Data content | Description |
|---|---|
| time | UTC time in Unix timestamp (seconds) |
| icao24 | Aircraft type designator |
| lat | Latitude of the aircraft position |
| lon | Longitude of the aircraft position |

| velocity | Ground speed in m/s |
|---|---|
| heading | Direction of movement as the clockwise angle from the geographic north |
| vertrate | Vertical speed in m/s |
| callsign | Callsign of the aircraft |
| onground | At surface position = true; **At airborne positions = false (only false condition is required)** |
| alert | Special indicators used in ATC |
| spi | Special indicators used in ATC |
| squawk | 4-digit octal number used by ATC and pilot represent for emergency condition |
| baroaltitude | Altitude in meters measured by barometer, the value will be slightly different from the value measured by GNSS sensor(geoaltitude), but baroaltitude always present in measurement |
| geoaltitude | Altitude in meters measured by GNSS sensor |
| lastposupdate | Time indicate the age of the position information, in Unix timestamp(second) |
| lastcontact | Time last receive signal from aircraft, time in Unix timestamp (second) |
| hours | Unix timestamp(second) indicating the hour |

To overcome the problem of obtaining undesired ADS-B data, a second attempt was made using a Python script to extract ADS-B data rather than using the open-source wrapper. The script is designed to extract only the data content required at specific times and from geographical areas defined by the researchers. This method also filters out records that have missing position information (e.g.: the barometric altitude) while querying ADS-B data in the OpenSky Network database. In this way, the data obtained will not need to go through the process of removing undesired data content in the next phase (data transformation). This approach, however, requires knowing the server address and the port number before establishing the shell connection. Information on this implementation can be found on the OpenSky Network website. In addition, it requires more time to extract the data because of the filtering process. The complete Python script can be found in Appendix B2. The 'pyopensky' wrapper has a similar capability to filter out undesired data content during query operation, which is achieved by passing SQL query to the wrapper. The result obtained is the same as the script presented in Appendix B2.

Using a similar approach to that applied to weather data, the size and time required to download ADS-B data varies depending on the selected time, geographical area as well as the network quality and the performance of the computer. An example is given in Figure 18 where the logs for downloading historical air traffic data from the OpenSky Network are compared with and without the filtering process being applied during the data extraction process. The ADS-B data in the example was collected on August 1st, 2020 from 15:00:00 to 15:01:00 UTC around the east coast of the United States of America.

```
-------------------  pyopensky wrapper  -------------------
ADS-B data time(UTC): 2020-08-01 15:00:00 to 2020-08-01 15:01:00
Geographical area: Latitude = [40, 42] Longitude=[-75,-72]
Data download duration: 4.0 minutes 57.386947 second
Total number of records found: 375033
Number of unique aircraft: 6594
File size: 63.148 MB
```

Obtained from wrapper to extract all data

```
--------------------  script  --------------------
ADS-B data time(UTC): 2020-08-01 15:00:00 to 2020-08-01 15:01:00
Geographical area: Latitude = [40, 42] Longitude=[-75,-72]
Data download duration: 22.0 minutes 13.535462 second
Total number of records found: 7121
Number of unique aircraft: 130
File size: 1.104 MB
```

Obtained from script to extract desired data

*Figure 18. ADS-B data download log comparison*

Figure 18 indicates that it is more efficient to download all ADS-B data available on the OpenSky Network server for a specific time and geographical location, but the information obtained will be need additional processing to remove undesired information. While applying the filter and using the script to download the ADS-B data takes more time to execute, the information obtained will not require additional processing.

Figure 19 presents snips of the ADS-B data files obtained using the two different methods. A comparison of the two data files shows that the data obtained from the script only contains the information required for the specified research, and there is no missing position information in the dataset. Some data records have missing aircraft callsign information. In the ADS-B data, an aircraft can be identified with its unique callsign or aircraft type designator, and for those records that are missing aircraft callsign, a cross-reference can be applied to the data file by searching the aircraft type designator; locating other records associated with the same aircraft; and using those records to identify the missing callsign information. In the case study that is aprt of this research, the aircraft type designator is used as the aircraft identification code, while the callsign is used as backup information. For this reason, missing callsign in the data file is disregarded as long as there is an aircraft type designator associated with the record.

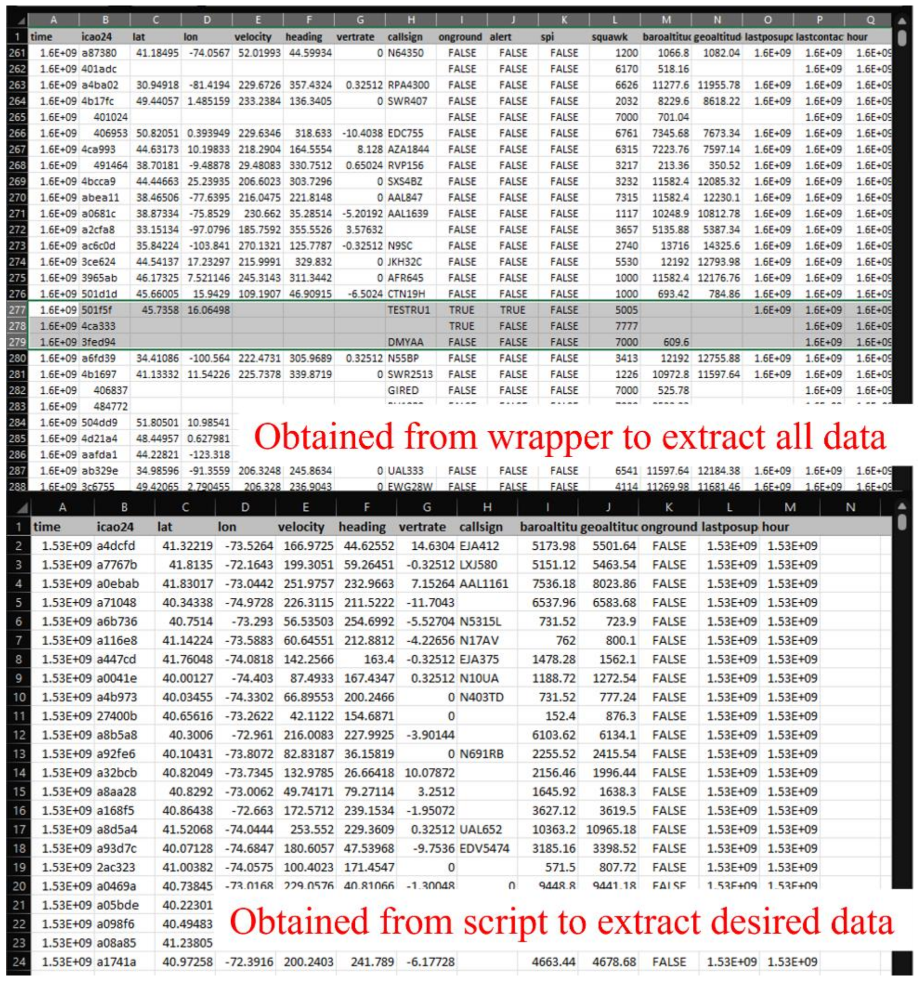


*Figure 19. ADS-B data file comparison*

In this section, the data extraction method used to extract weather radar and air traffic data has been presented. The selection of the data extraction method depends on how the data is presented at the source as well as what then user requires for the research. The importance of the data extraction phase is to extract raw data from selected data sources that meet the research requirement without unnecessary or missing data components. The next section will present the implementation of data transformation on the raw data downloaded in this case study.

### 4.3.2 Data transformation

There are three processes in the data transformation phase as shown in Figure 11; decoding raw data, filtering, and data standardization. This section will present the implementation of each of these processes in the context of the case study.

#### 4.3.2.1 Decoding raw data

The ADS-B data obtained from the OpenSky Network historical database has already been decoded but because all Impala-shell query results are returned in plain text format, the downloaded data must be converted to a CSV file. This was performed at the time the data was extracted from the OpenSky Network database, and the implementation of the operation can be found in the data extraction Python script provided in Appendix B2.

The raw radar data obtained from NEXRAD requires decoding in order to provide a readable format for the user. Raw radar data downloaded from the NCEI AWS S3 bucket can be decoded using special software, and the NCEI offers a list of free decoders for various programming languages on its website.  In this case study, the NOAA's WCT distributed from NCEI was selected. The WCT is a decoding software that is officially released by the NCEI and using WCT for the case study ensures the integrity of the decoded data. A Python script was developed to run the WCT through command lines. The complete script can be found in Appendix C, which was used to run the execution of decoding raw radar files, the exporting of data to CSV files.

#### 4.3.2.2 Filtering data content

After the data is decoded and converted to a CSV file, the next step is to filter out data components that are not required for the research. In the context of the case study described here, this step only applies to the weather radar data because the ADS-B data was filtered as it was extracted from the OpenSky Network database as described in Section 4.3.1.2.

Tables 8 and 9 show the data components of the weather radar data after the decoding process. Only the data content highlighted in green has been requested by the researcher, and the data content highlighted in red needs to be removed. This process is performed as part of the Python

script developed for extracting the weather radar data from the NCEI Amazon S3 bucket, and the detailed implementation can be found in the Python script provided in Appendix C.

*Table 8. Raw weather radar data (part 1)*

| **Raw weather data (part 1)** | | | | | |
|---|---|---|---|---|---|
| Data content | Sweep | sweepTime | elevAngle | value | radialAng |
| Description and data unit | Radar sweep (0-360 degree) | Time of the sweep, Zulu time, UTC (YYYY-MM-DDT Hh:mm:ssZ) | The tilt angle between the horizontal plane and the line of sight of the current sweep (degree) | Base reflectivity value (dBZ) | Azimuth angle in the radar's polar coordinate system, where 0 is north (degree) |

*Table 9. Raw weather radar data (part 2)*

| **Raw weather data (part 2)** | | | | | |
|---|---|---|---|---|---|
| Data content | surfaceRan | heightRel | heightASL | latitude | longitude |
| Description and data unit | Range of radar site(m) | Height relative to radar(m) | Height above sea level (m) | Latitude of the base reflectivity data point(Angular degree) | Longitude of the base reflectivity data point (Angular degree) |

### 4.3.2.3 Data standardization

Once all datasets contain only the required data, the remaining process in the data transformation phase is to standardize the naming conventions and data units for common data contents. Tables 10, 11 and 12 present the data content from the two datasets used for the case study. The columns highlighted in blue represent common data components shared by the two datasets, where time and geographical information are the common data contents in this case. A standardized naming convention and data unit must be defined and implemented to be compatible with the design of the database schema described in Section 4.2.

*Table 10. Data content of weather radar dataset*

| Filtered weather radar data | | | | | |
|---|---|---|---|---|---|
| Data content | sweepTime | value | heightASL | latitude | longitude |
| Description and data unit | Time of the sweep, Zulu time, UTC (YYYY-MM-DDT Hh:mm:ssZ) | Base reflectivity value (dBZ) | Height above sea level (m) | Latitude of the base reflectivity data point (Angular degree) | Longitude of the base reflectivity data point (Angular degree) |

The weather radar dataset uses the label 'sweepTime' to identify the time when the data is collected, while the ADS-B data obtained from the OpenSky Network uses the label "time". Although both datasets use Coordinated Universal Time (UTC) standard, one dataset presents time as YYYY-MM-DDThh:mm:ssZ, whereas the other dataset presents time data in seconds in the Unix time format. In this case study, the 'sweepTime' label from the weather radar dataset is renamed to 'time' to keep the naming consistency in both datasets. The Unix time format is selected to represent time in both datasets because the format makes it easy for computers to store, manage and compare data. The implementation of this operation can be found in the Python script provided in Appendix C.

*Table 11. Data content of ADS-B dataset (part1)*

| ADS-B data (part 1) | | | | | | |
|---|---|---|---|---|---|---|
| Data content | time | icao24 | lat | lon | velocity | heading |
| Description and data unit | UTC time of the data captured (Unix time) | Icao code of the aircraft (text string) | Latitude of the aircraft position (Angular degree) | Longitude of the aircraft position (Angular degree) | Ground speed (m/s) | Direction of the movement as the clockwise angle from the geographic north (Angular degree) |

*Table 12. Data content of ADS-B dataset (part2)*

| ADS-B data (part 2) | | | | | |
|---|---|---|---|---|---|
| Data content | vertrate | callsign | baroaltitude | geoaltitude | lastposupdate |
| Description and data unit | Vertical speed (m/s) | Callsign of the aircraft (text string) | Aircraft altitude measured by barometer (m) | Aircraft altitude measured by GNSS sensor (m) | Time indicate the age of the position information (Unix time) |

The geographical information data content also requires the standardization of both data units and naming convention. Although the angular degree is used to represent the geographical coordinate in both datasets, the naming convention between the two is not the same. The labels 'lat' and 'lon' in the ADS-B dataset are renamed as 'latitude' and 'longitude' to keep the naming consistent between both datasets. Tables 13, 14 and 15 show the data content of the two datasets after the data standardization process.

*Table 13. Standardized weather radar data*

| Standardized weather radar data | | | | | |
|---|---|---|---|---|---|
| Data content | time | latitude | longitude | value | heightASL |
| Data unit | Unix time | Angular degree | Angular degree | dBZ | m |

*Table 14. Standardized ADS-B data (part 1)*

| ADS-B data (part 1) | | | | | | |
|---|---|---|---|---|---|---|
| Data content | time | latitude | longitude | icao24 | velocity | heading |
| Data unit | Unix time | Angular degree | Angular degree | Text string | m/s | Angular degree |

*Table 15. Standardized ADS-B data (part 2)*

| **ADS-B data (part 2)** | | | | | |
|---|---|---|---|---|---|
| Data content | vertrate | callsign | baroaltitude | geoaltitude | lastposupdate |
| Data unit | m/s | Text string | m | m | Unix time |

Metres are used in both datasets as the unit for altitude information, but these data contents are all kept in their own form as they are collected based on different standards. The altitude information in the weather dataset is the height of the radar antenna above sea level, whereas, in the ADS-B dataset, there are barometric altitudes and geographical altitudes. According to the description provided by the OpenSky Network, the ADS-B data provided will almost always have the barometric altitude information, but in case of missing information, the geographical altitude information of each ADS-B data is extracted from the OpenSky Network historical database. A new data column will be created to manage the altitude information while loading the data to the new database for storage and will be presented in Section 4.3.3.

The ADS-B data used in this case study require little modification, and only the semantics and scientific units are edited for shared data content. The size of the ADS-B data file is mainly determined by when and where the data was captured and is not significantly impacted by the data standardization process. Table 16 provides an example of file size before and after the process for ADS-B data collected around the New York area on June 1st, 2020 from 00:00 - 00:03 UTC.

*Table 16. ADS-B data file size comparison*

| **Description** | **File size** |
|---|---|
| Original data file | 5,569 KB (5.569 MB) |
| Standardized data file | 5,664 KB (5.664 MB) |

The weather radar data requires a complete data transformation process from decoding the raw data to data standardization. During this process, the file size of the weather radar data file changes significantly. Table 17 provides an example of how the file size changes for 10 minutes of weather radar information collected at the weather radar KOKX on August 1st, 2020 around 00:14 UTC.

*Table 17. Weather data file size comparison*

| Description | File size |
|---|---|
| Raw data file | 2,412 KB (2.412 MB) |
| Decoded data file | 16,320 KB (16.32 MB) |
| Filtered and standardized data file | 9,482 KB (9.482 MB) |

### 4.3.3 Loading data to storage

Once all the datasets are cleaned and standardized, the last step in the data ETL process is to load the data to the database for storage. In this thesis, the database is built with SQLite. The database schema shown in Figure 15 is implemented to store the required weather radar and ADS-B data.

In the data transformation phase, the two datasets are converted to the CSV file format for processing. In Python, there is a library extension that can convert CSV files to local database files. The library extensions used to assist in this process are named 'pandas' and 'sqlite3'. Pandas is a Python library that was developed for data manipulation and analysis, while sqlite3 is used for creating and managing database files. Appendix D contains the complete Python script for loading the cleaned and standardized weather radar data and ADS-B data to the research database.

In the database schema shown in Figure 15, the *geo_BLOB* column is a spatial data object created by using the latitude, longitude, and altitude of the current data row. This is achieved by using a spatial extension of SQLite named Spatialite [96]. Figure 20 presents an example of a database file obtained from the implementation. The table highlighted in blue corresponds to the four data tables presented in the database schema shown in Figure 15. The other tables shown on the list come with the Spatialite extension. The example presented in Figure 20 contains 5 minutes of ADS-B data and 10 minutes of weather radar data collected on August 1$^{st}$, 2020 at around 00:14 UTC. The size of this database is around 58 MB. The size of the database is bigger than the sum of the two original datasets due to adding the Spatialite extension. However, the Spatialite extension is important for this case study, as it is designed to support the management and querying of data from geodatabases.

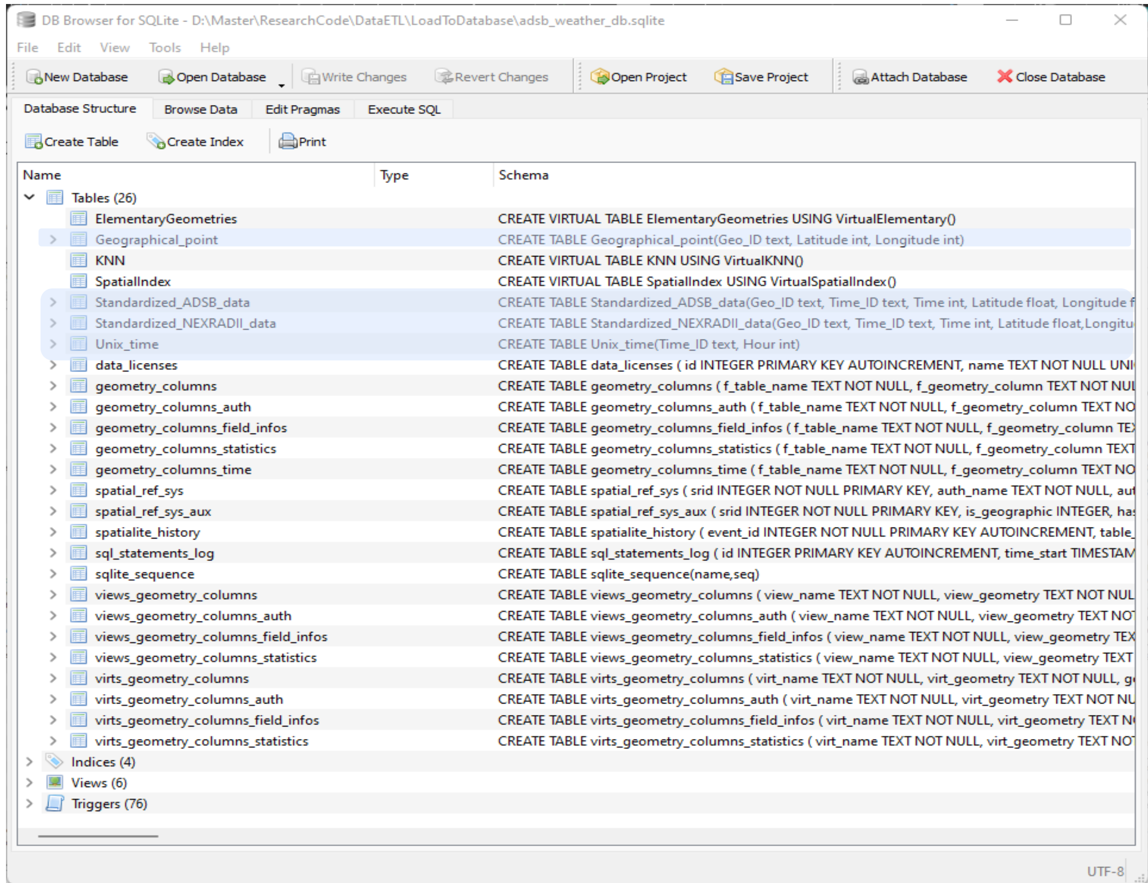


*Figure 20. An overview of database*

## 4.4 GUI development for data visualization

This section presents a prototype of the data visualization tool developed for the case study. The visualization tool must meet the following requirements:

1. The tool should be able to create simulated air traffic scenarios by using any selected data from the database.
2. The tool should be able to generate a 2D and a 3D visualization in the simulated airspace environment.

3. The tool should be able to display the collection date and time of the data in use in the simulated airspace environment.
4. The tool should be able to generate animation intended to visualize how the air traffic and weather change with respect to time.
5. The tool should have functions that allow users to visualize and extract data on the interaction between different airspace entities within the simulated flight scenario.
6. The tool should be able to be used with any electronic device.

Figure 21 shows the first prototype developed as part of this thesis research to load and visualize the ADS-B data from the database. It was developed using Python with BaseMap [97], one of its library extensions.

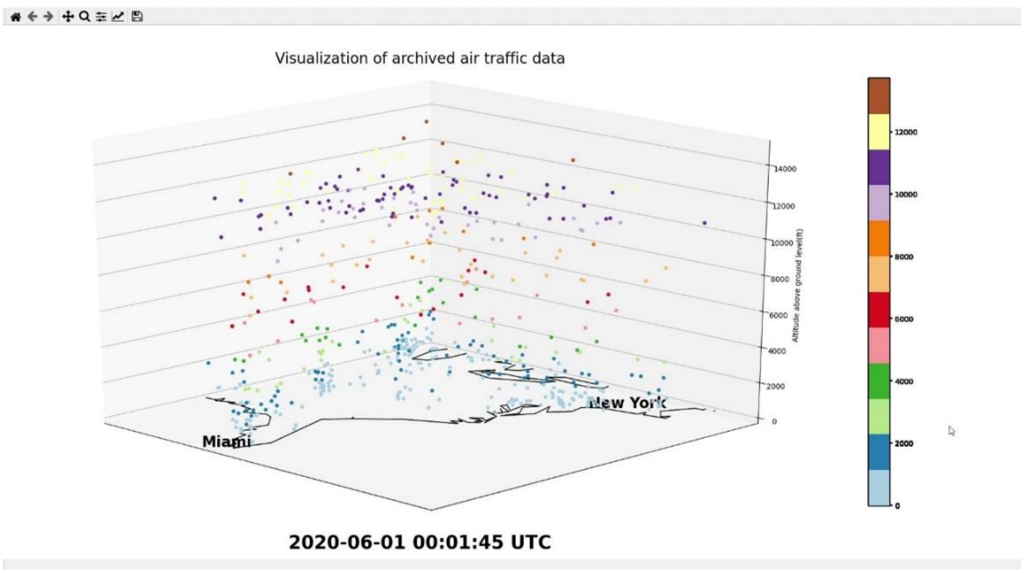


*Figure 21. ADS-B data representation in Prototype #1*

In Prototype #1, the simulated airspace environment is presented in a three-dimensional space. Video 1, accessible through the link provided below, is a 2-minute archived air traffic playback using Prototype #1. The ADS-B data loaded to Prototype #1 was captured on June 1st, 2020 around 00:01:00 UTC.

Video 1 – Prototype #1 demonstration: https://www.youtube.com/watch?v=9H4eG0KYgTk

Figure 21 is a snapshot taken Video 1. The horizontal plane represents the geographic area around New York and Miami, and the vertical axis represents altitude above ground level. The three-dimensional space is filled with coloured dots each of which represents an individual aircraft from the ADS-B data in the database. The colour scale of the dots represents each aircraft's altitude.

Prototype #1 provides an intuitive visualization of how air traffic is distributed in a specific geographical area during a selected time period. However, it fails to provide additional information used to create the simulated airspace environment such as i) the callsign, the speed, and the heading of the aircraft, ii) the airspace infrastructure, and iii) the terrain in that area. Furthermore, this prototype does not provide a user interface for interacting with the data loaded into the simulated environment.

Prototype #2 includes a user interface and was developed in Python using the library extensions BaseMap [97] and wxPython [98]. Figure 22 shows the interface for Prototype #2.

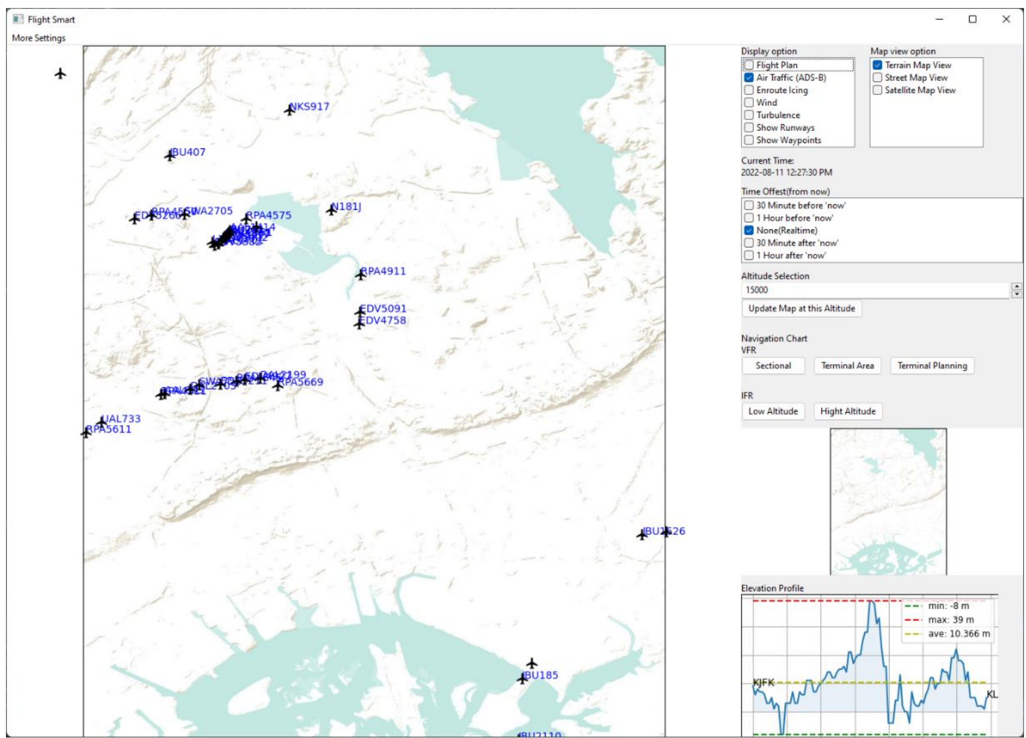


*Figure 22. Interface of Prototype #2*

The left side of the interface shows the geographical area being simulated (in this case the New York city area). The ADS-B data is overlaid on a 3-D topographical map using an aircraft icon associated with the ICAO code of the aircraft. User control units are located on the right side of the interface and can be used for interacting with the data displayed on the map. The disadvantages associated with prototype #2 include: i) the ADS-B data can only be presented in the top view, which makes it impossible to visualize the vertical separation between aircraft; and ii) Python is used to develop the tool and it is only compatible with desktop computers and not other electronic devices such as cellphone or tablet.

Prototype #3 is a web-based data visualization tool that can be used on any kind of electronic device. Prototype #3 was developed in Javascript using Mapbox studio [94], which provides custom online maps for websites and applications. Mapbox studio has built-in libraries for terrain and public transportation including airport and airspace infrastructure, and this information is presented in the form of data layers. Figure 23 shows how Mapbox studio is used to assist in generating the visual representation of the airspace environment.

In Figure 23, the six blue data layers are generated by using the information available in Mapbox studio. The visual representation of the simulated airspace environment is created by overlaying these data layers on top of one another. This is done using the online editing tool of the Mapbox studio, and an access token is generated. The access token is used in the Javascript code for developing Prototype #3 to ensure access to the visual representation of the simulated airspace environment at all times. In Figure 23, the weather and air traffic data layers colored in green are generated by using the information available in the database. As a consequence, Prototype #3 is capable of generating the visual representation of different flight scenarios by overlaying different weather and air traffic information on top of the simulated airspace environment.

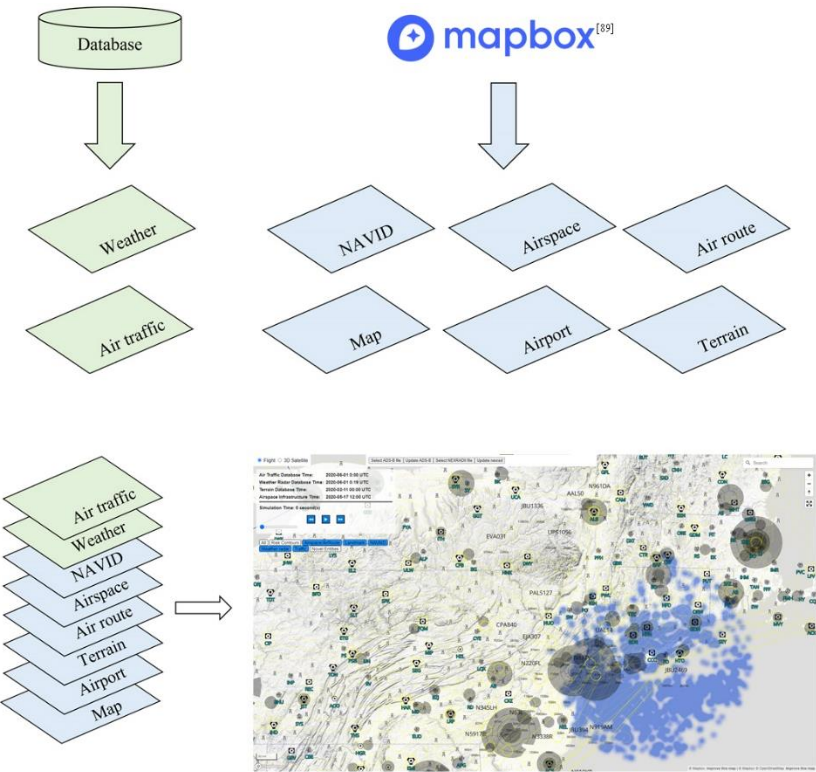


*Figure 23. Data layers in Prototype #3*

Figure 24 provides an example of Prototype #3 loaded with a visual representation of ADS-B aircraft and weather data. Five primary features are provided to users of Prototype #3 for interacting with the map or the data layers:

1. Two toggle switches for map type selection.
2. An information box shows the date and time for each of the data layers currently loaded on the map.
3. File selection and update buttons for modifying the weather data layer and air traffic data layer.
4. A media control panel for simulation playback.
5. On/Off buttons for controlling the visibility of each data layer.

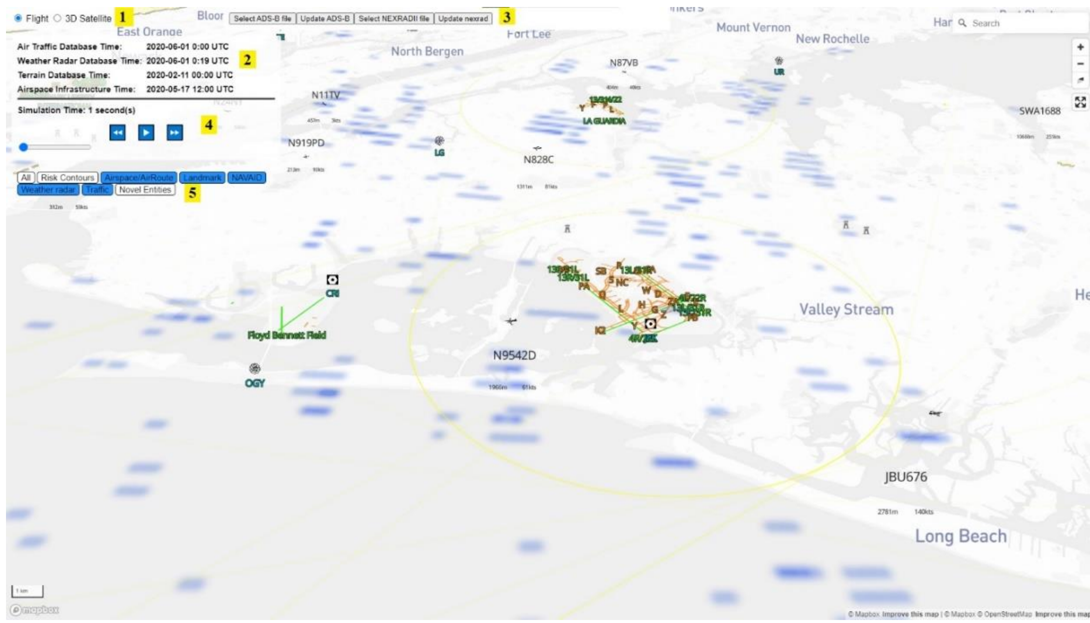


*Figure 24. The interface of Prototype #3 ('Flight' map style)*

There are two map styles available in Prototype #3, and the map shown in Figure 24 is the 'Flight' map style. Users can clearly see each layer of data on this monochrome map because it provides a clear background for overlaying different data layers. The other of available map style is the 'Satellite' map. Figure 25 shows the same scenario as Figure 24 but in the 'Satellite' map. Users can choose a map style by using the toggle switch on the top left corner identified with the number '1' in Figure 24.

Prototype #3 presents data by creating data layers and overlaying them. The data layers are graphical representations of the original data, and the prototype is designed in such a way that it is capable of loading data from any date and time as long as the data are available in the database. The capture date and time of each of the data layers shown on the map can be found in the information box denoted by the number '2' in Figure 24. The example presented in Figure 24 was created using the ADS-B and weather radar data collected on June 1$^{st}$, 2020 around 00:00 UTC, where the blue rendering represents the weather and the 3D plane model with accompanying ICAO code, ground speed, and altitude represents the air traffic. The yellow line on the map is the air route from the airspace data layer and shows the airspace boundary and the airways around the JFK airport New York. The airport infrastructure runways and taxiways are highlighted in green and orange.

The weather data layer and the air traffic in Prototype #3 can be modified at any time by loading the available data from the database. The buttons designed for selecting and updating the data files are denoted by the number ‘3’ in Figure 24.

The media control box denoted by the number ‘4’ in Figure 24 is used for controlling the playback of the animation. ‘Simulation time’ indicates the time passed in the simulated airspace environment, and users can use the media control button to enact and change or adjust the animation process. Once the data is loaded, clicking the play button will allow the tool to animate the aircraft and weather systems and observe a visual representation of the flow of air traffic. The animation can be stopped at any time, fast forwarded and rewound.

To prevent potentially confusing amounts of presented information, each data layer on the map can be inactivated. The buttons designed for this function are identified by '5' in Figure 24, with the name tags indicating the data layer they control. When the background colour of the button is blue, it means the data layer is activated, and when the background colour is white the data layer is inactivated and does not display on the map.

Of the three prototypes developed to allow researchers to understand, visualize and interact with data, prototype #3 is the one that satisfies all the requirements defined at the beginning of the project. Video 2, accessible through the link provided below, shows an overview of each data layers on Prototype #3 and demonstrate the capability of the control units. The next section will present the process of verifying the database design for the case study by comparing the features of Prototype #3 with the defined needs of the research case.

Video 2 - Prototype #3 overview: https://youtu.be/LSB7ex49N-U

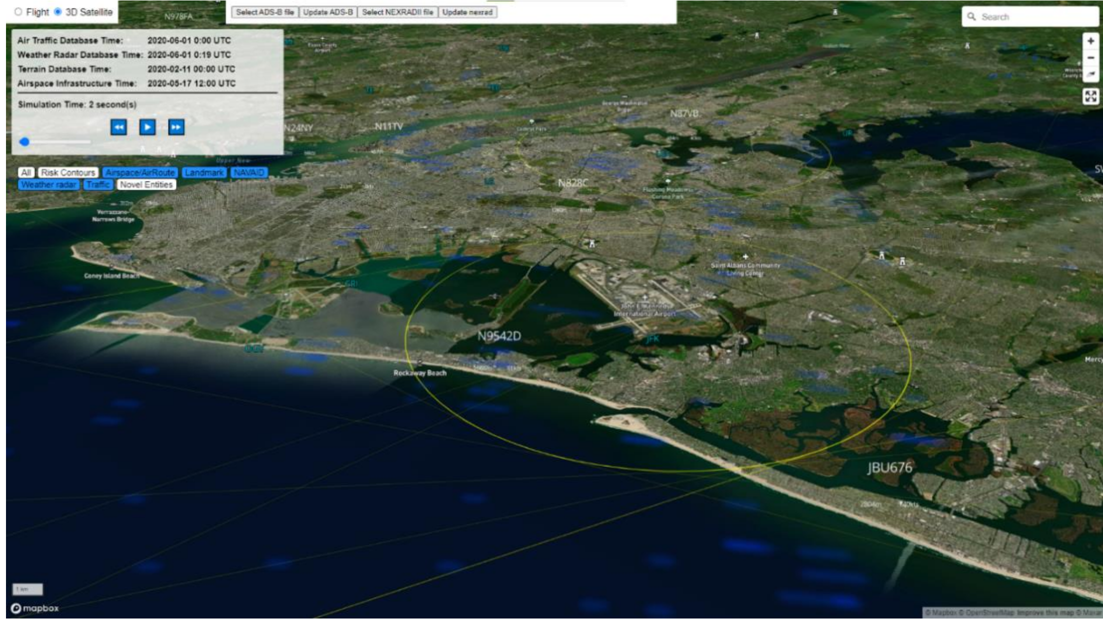


*Figure 25. The interface of Prototype #3 ('Satellite' map style)*

## 4.5 Verification with test case

The purpose of this section is to verify that the Prototype #3 database obtained as described in Sections 4.1 through 4.4 satisfies the research requirements defined for the case study:

1. A variety of flight scenarios can be generated with the information available in the database.
2. The visualization tool allows users to visualize and analyze interactions between the following entities:
    1. Air traffic
    2. Air space and airport infrastructure
    3. Ground obstacle
    4. Terrain
    5. Weather
3. The tool can generate the visual representation of the selected simulated flight scenario efficiently.

In addition to these general criteria, the researcher specifically wanted to analyse the impact of introducing a novel air vehicle into an existing airspace environment. For this reason, the prototype is validated with the addition of another layer representing the novel air vehicle using data provided by the researcher for i) a predefined flight route; ii) an actual flight route; and iii) the heading of the novel air vehicle.

Figure 26 and Figure 27 are screenshots of the visual tool when introducing the novel air vehicle into the simulated airspace environment presented in Figure 21. The information box located at the bottom left corner shows information about the novel air vehicle. The predefined flight route in this case is from JFK to LGA, depicted by the blue line shown in both figures. Figure 26 shows how the given flight route appears from the bird's eye view while Figure 27 present the same scenario in a 3D view.

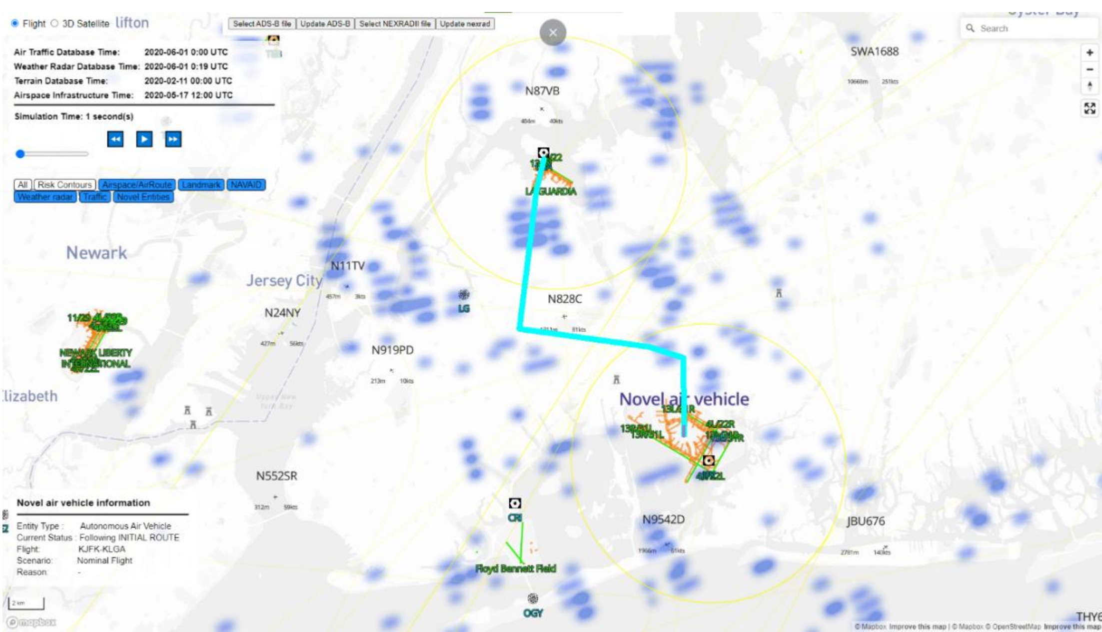


*Figure 26. Introduce novel air vehicle to Prototype #3 (bird's eye view)*

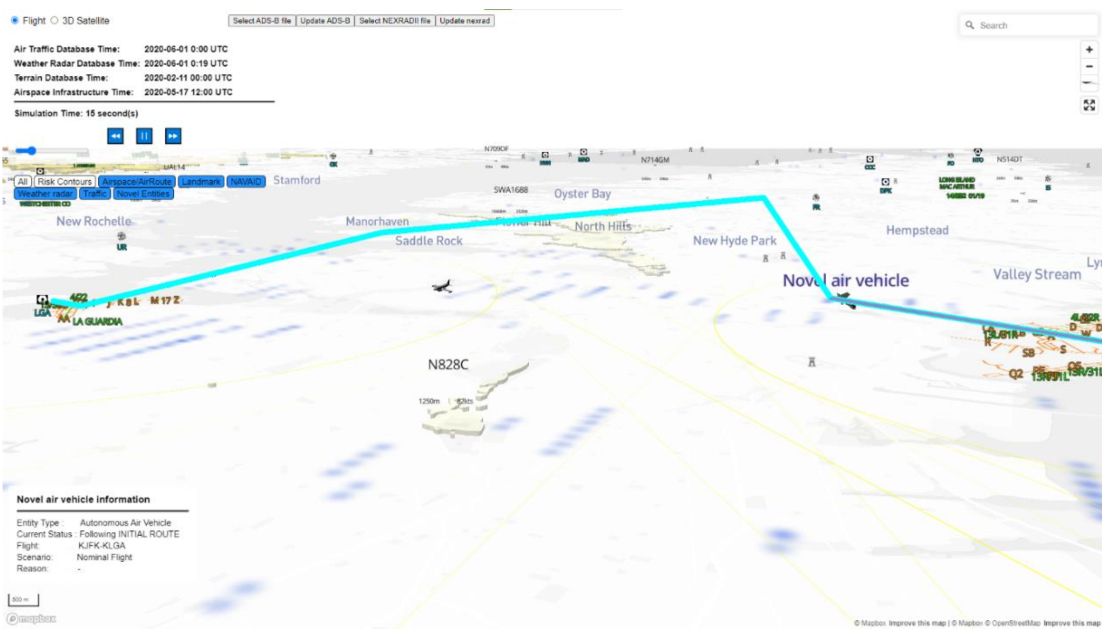


*Figure 27. Introduce novel air vehicle to Prototype #3(3D view)*

To distinguish the novel air vehicle from other ADS-B aircraft, the 3-D image representation is unique and a text label 'Novel air vehicle' is displayed alongside aircraft. The blue line indicates the flight route, and the update frequency of the novel air vehicle and the ADS-B aircraft is synchronized in order to animate the scenario and any possible interactions in “real-life”. Video 3 demonstrate the simulated flight scenario present in Figure 26 and Figure 27.

Video 3 – Prototype #3 animating simulated flight scenario: https://youtu.be/xO1qfkIHghE

Prototype #3 is capable of adapting to changes in research requirements because it treats all the data separately and presents them in separate data layers. If a new or additional type of data is required, it can be converted to a data layer and overlaid on top of the existing layers. Figure 28 below shows the process of modifying Prototype #3 to add the novel air vehicle data layer.

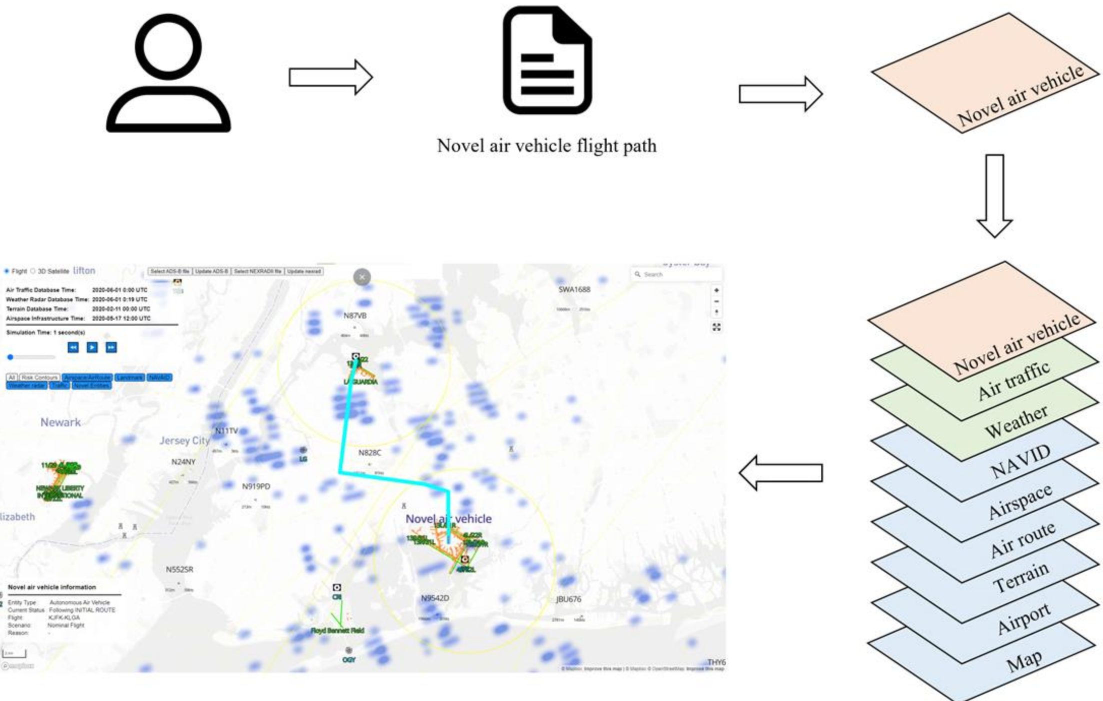


*Figure 28. Adding new data layer to Prototype #3*

The clean and organized data in the database makes it possible to create novel simulated airspace environments by combining historical data captured at different times. Figure 28 is an example of generating an imaginary but true-to-life flight scenario in Prototype #3. For example, the simulated airspace environment shown in Figure 26 combines data captured at different times; the weather radar data was collected on 1st October 2020 at 00:03 UTC whereas the air traffic data was captured on 1st June 2020 at 00:00 UTC.

The simulated airspace environment shown in Figure 29 is made by combining real-world data from different days and times to provide a wide range of flight scenarios for research purposes. The capability to combine layers from different times allows the researchers to combine and study potentially hazardous air traffic and weather scenarios that would be too dangerous to attempt in real time.

*Figure 29. Generate novel simulate airspace environment*

Should the research results obtained using combined scenarios create a need for additional data from other geographic locations, different weather conditions or traffic patterns, this can be readily achieved by repeating the process presented in Section 4.3 and storing the new data to the existing database.

# 5. Conclusion

Data-driven aviation research is becoming increasingly popular as researchers realize the value and potential of using available data, but many users face challenges related to the large amounts of data and the variety of formats that require pre-processing before being used. In response, some researchers have created very large databases containing cleaned and neatly organized data meant to suit the needs of a large variety of research projects. There remain, however, situations where the information is too general to suit specific research needs; the amount of data in the database is unmanageable for smaller computer systems; or the size of the database makes it difficult to navigate, find and extract information related to detailed requirements.

In this thesis, a methodology is presented for the design and development of a data repository for specific aerospace applications. The methodology presented is a systematic approach that can be applied to many kinds of data-driven research. An example is presented where archived historical weather radar and air traffic data is used to create a data repository in support of research requiring a simulated airspace environment. A case study is presented to illustrate the steps in the process of selecting, extracting, transforming, and loading the data into a database. The method is validated by using the data to create a variety of flight scenarios using a system of layers; and a visualisation tool is presented that allows researchers and other participants to better understand how the data is interacting. Data from the prototype database can be used to develop a variety of simulated flight scenarios to support the research in the case study.

The approach presented in this thesis is not capable of handling real-time data streams. This research focuses on leveraging historical data, and the proposed method can be achieved by using a normal desktop computer. Applying the proposed method to a real-time data stream would be problematic because of the large amounts of incoming data as well as problems with missing information and inconsistent scientific units.

This research clearly illustrates the usefulness of developing a compact database based on specific research requirements for the purposes of supporting data-driven research, but it also raises the question of data integrity in the selected data source. The methodology presented in this thesis does not include sustainable solutions for resolving problems related to incomplete or inaccurate information obtained from the selected data source. The problem is acknowledged, and the temporary solution presented in this thesis is to select a data source from a trustworthy organization.

Future work for improving the methodology presented in this thesis could be developing a method to filter out inaccurate information or to identify missing information by cross referencing the same types of data collected from different data sources.

# 6. References

[1] H. K. Ng, B. Sridhar, and S. Grabbe, 'Optimizing Aircraft Trajectories with Multiple Cruise Altitudes in the Presence of Winds', *J. Aerosp. Inf. Syst.*, vol. 11, no. 1, pp. 35–47, Jan. 2014, doi: 10.2514/1.I010084.

[2] D. P. Thipphavong, C. A. Schultz, A. G. Lee, and S. H. Chan, 'Adaptive Algorithm to Improve Trajectory Prediction Accuracy of Climbing Aircraft', *J. Guid. Control Dyn.*, vol. 36, no. 1, pp. 15–24, Jan. 2013, doi: 10.2514/1.58508.

[3] K. (May) Ren, A. M. Kim, and K. Kuhn, 'Exploration of the Evolution of Airport Ground Delay Programs', *Transp. Res. Rec. J. Transp. Res. Board*, vol. 2672, no. 23, pp. 71–81, Dec. 2018, doi: 10.1177/0361198118782272.

[4] M. Alderighi and A. A. Gaggero, 'Flight cancellations and airline alliances: Empirical evidence from Europe', *Transp. Res. Part E Logist. Transp. Rev.*, vol. 116, pp. 90–101, Aug. 2018, doi: 10.1016/j.tre.2018.05.008.

[5] K. D. Bilimoria, B. Sridhar, S. R. Grabbe, G. B. Chatterji, and K. S. Sheth, 'FACET: Future ATM Concepts Evaluation Tool', *Air Traffic Control Q.*, vol. 9, no. 1, pp. 1–20, Jan. 2001, doi: 10.2514/atcq.9.1.1.

[6] K. Sheth *et al.*, 'Evolution of an Air Traffic Simulation Testbed into an Operational Tool', p. 12.

[7] S. George *et al.*, 'Build 8 of the Airspace Concept Evaluation System', in *AIAA Modeling and Simulation Technologies Conference*, Portland, Oregon, Aug. 2011. doi: 10.2514/6.2011-6373.

[8] B. Sridhar, G. Chatterji, S. Grabbe, and K. Sheth, 'Integration of Traffic Flow Management Decisions', in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, Monterey, California, Aug. 2002. doi: 10.2514/6.2002-5014.

[9] K. D. Bilimoria, S. R. Grabbe, K. S. Sheth, and H. Q. Lee, 'Performance Evaluation of Airborne Separation Assurance for Free Flight', *Air Traffic Control Q.*, vol. 11, no. 2, pp. 85–102, Apr. 2003, doi: 10.2514/atcq.11.2.85.

[10] K. S. Sheth, T. S. Islam, and P. H. Kopardekar, 'Analysis of airspace tube structures', in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, St. Paul, MN, USA, Oct. 2008, p. 3.C.2-1-3.C.2-10. doi: 10.1109/DASC.2008.4702805.

[11] C. Chen *et al.*, 'Defining Well Clear Separation for Unmanned Aircraft Systems Operating with Noncooperative Aircraft', in *AIAA Aviation 2019 Forum*, Dallas, Texas, Jun. 2019. doi: 10.2514/6.2019-3512.

[12] G. Satapathy, N. Nigam, and Y. Zhang, 'Sensitivity of Efficient Descent Advisor (EDA) Performance to Trajectory Prediction (TP) Errors', in *AIAA Guidance, Navigation, and Control Conference*, Portland,

Oregon, Aug. 2011. doi: 10.2514/6.2011-6663.
[13] S. Zelinski, 'Validating The Airspace Concept Evaluation System Using Real World Data', in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, San Francisco, California, Aug. 2005. doi: 10.2514/6.2005-6491.
[14] S. Zelinski and L. Meyn, 'Validating the Airspace Concept Evaluation System for Different Weather Days', in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Keystone, Colorado, Aug. 2006. doi: 10.2514/6.2006-6115.
[15] H. Erzberger, T. Nikoleris, R. A. Paielli, and Y.-C. Chu, 'Algorithms for control of arrival and departure traffic in terminal airspace', *Proc. Inst. Mech. Eng. Part G J. Aerosp. Eng.*, vol. 230, no. 9, pp. 1762–1779, Jul. 2016, doi: 10.1177/0954410016629499.
[16] J. Smith, N. Guerreiro, J. Viken, S. Dollyhigh, and J. Fenbert, 'Meeting Air Transportation Demand in 2025 by Using Larger Aircraft and Alternative Routing to Complement NextGen Operational Improvements', in *10th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, Fort Worth, Texas, Sep. 2010. doi: 10.2514/6.2010-9107.
[17] V. N. Gudivada, R. Baeza-Yates, and V. V. Raghavan, 'Big Data: Promises and Problems', *Computer*, vol. 48, no. 3, pp. 20–23, Mar. 2015, doi: 10.1109/MC.2015.62.
[18] NIST Big Data Public Working Group Definitions and Taxonomies Subgroup, 'NIST Big Data Interoperability Framework: Volume 1, Definitions', National Institute of Standards and Technology, NIST SP 1500-1, Oct. 2015. doi: 10.6028/NIST.SP.1500-1.
[19] Doug Laney, '3D data management : controlling data volume, velocity and variety', *META Group Res*, vol. 949, no. February 2001, pp. 4–4, 2001.
[20] V. Shobana and N. Kumar, 'Big data - A review', *Int. J. Appl. Eng. Res.*, vol. 10, no. 55, pp. 1294–1298, 2013, doi: 10.26634/jit.6.1.13507.
[21] M. Minelli, M. Chambers, and A. Dhiraj, *Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Business*. 2013. doi: 10.1002/9781118562260.
[22] A. K. Bhadani and D. Jothimani, 'Big data: Challenges, opportunities, and realities', *Eff. Big Data Manag. Oppor. Implement.*, pp. 1–24, 2016, doi: 10.4018/978-1-5225-0182-4.ch001.
[23] Z. Bai and X. Bai, 'Sports Big Data: Management, Analysis, Applications, and Challenges', *Complexity*, vol. 2021, pp. 1–11, Jan. 2021, doi: 10.1155/2021/6676297.
[24] B. Han, Z. Chen, C. Liu, and M. Shang, 'Design and Implementation of Big Data Management Platform for Android Applications', in *Proceedings of the 2020 3rd International Conference on Big Data Technologies*, Qingdao China, Sep. 2020, pp. 36–40. doi: 10.1145/3422713.3422715.
[25] J. Chen, S. He, and X. Li, 'A Study of Big Data Application in Agriculture', *J. Phys. Conf. Ser.*, vol. 1757, no. 1, p. 012107, Jan. 2021, doi: 10.1088/1742-6596/1757/1/012107.
[26] A. A. Munshi and A. Alhindi, 'Big Data Platform for Educational Analytics', *IEEE Access*, vol. 9, pp.

52883–52890, 2021, doi: 10.1109/ACCESS.2021.3070737.
[27] K. Xiong, ‘Research on Innovation of Automobile Marketing Mode Based on Big Data Marketing’, in *2020 International Conference on Big Data and Social Sciences (ICBDSS)*, Xi’an, China, Aug. 2020, pp. 72–75. doi: 10.1109/ICBDSS51270.2020.00024.
[28] A. Su, ‘Tourism Marketing Innovation Management Model Based on Big Data’, *J. Phys. Conf. Ser.*, vol. 1744, no. 4, p. 042141, Feb. 2021, doi: 10.1088/1742-6596/1744/4/042141.
[29] L. Zhe, L. Xueyan, and T. Huan, ‘Research On The Architecture And Strategy Of Luxury brands Marketing Service Design Model From The Perspective Of Big Data’, *E3S Web Conf.*, vol. 179, p. 02014, 2020, doi: 10.1051/e3sconf/202017902014.
[30] D. Gupta and R. Rani, ‘A study of big data evolution and research challenges’, *J. Inf. Sci.*, vol. 45, no. 3, pp. 322–340, Jun. 2019, doi: 10.1177/0165551518789880.
[31] M. Aparicio and C. J. Costa, ‘Data Visualization’, *Commun. Des. Q.*, vol. 3, no. 1, pp. 7–11, Nov. 2014, doi: 10.1145/2721882.2721883.
[32] M. T. Rodríguez, S. Nunes, and T. Devezas, ‘Telling Stories with Data Visualization’, in *Proceedings of the 2015 Workshop on Narrative & Hypertext - NHT ’15*, Guzelyurt, Northern Cyprus, 2015, pp. 7–11. doi: 10.1145/2804565.2804567.
[33] M. Mani and S. Fei, ‘Effective Big Data Visualization’, in *Proceedings of the 21st International Database Engineering & Applications Symposium on - IDEAS 2017*, Bristol, United Kingdom, 2017, pp. 298–303. doi: 10.1145/3105831.3105857.
[34] R. Agrawal, A. Kadadi, X. Dai, and F. Andres, ‘Challenges and opportunities with big data visualization’, in *Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems*, Caraguatatuba Brazil, Oct. 2015, pp. 169–173. doi: 10.1145/2857218.2857256.
[35] D. M. Eldin, A. E. Hassanien, and E. E. Hassanien, ‘Challenges of Big Data Visualization in Internet-of-Things Environments’, in *International Conference on Innovative Computing and Communications*, vol. 1087, A. Khanna, D. Gupta, S. Bhattacharyya, V. Snasel, J. Platos, and A. E. Hassanien, Eds. Singapore: Springer Singapore, 2020, pp. 873–885. doi: 10.1007/978-981-15-1286-5_76.
[36] S. Sachchidanand and S. Nirmala, ‘Big data analytics’, in *Resonance*, 2012, vol. 21, pp. 695–716. doi: 10.1007/s12045-016-0376-7.
[37] ‘Guide to Laptop Storage Drives’, *Lifewire*. https://www.lifewire.com/laptop-storage-drives-guide-833445 (accessed Dec. 13, 2022).
[38] L. Mearian, ‘World’s data will grow by 50X in next decade, IDC study predicts’, *Computerworld*, Jun. 28, 2011. https://www.computerworld.com/article/2509588/world-s-data-will-grow-by-50x-in-next-decade--idc-study-predicts.html (accessed Dec. 13, 2022).
[39] Z. D. Stephens *et al.*, ‘Big Data: Astronomical or Genomical?’, *PLOS Biol.*, vol. 13, no. 7, p. e1002195,

Jul. 2015, doi: 10.1371/journal.pbio.1002195.
[40] B. Diène, J. J. P. C. Rodrigues, O. Diallo, E. H. M. Ndoye, and V. V. Korotaev, ‘Data management techniques for Internet of Things’, *Mech. Syst. Signal Process.*, vol. 138, p. 106564, Apr. 2020, doi: 10.1016/j.ymssp.2019.106564.
[41] N. Saranya, R. Brindha, N. Aishwariya, R. Kokila, P. Matheswaran, and P. Poongavi, ‘Data Migration using ETL Workflow’, in *2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS)*, Coimbatore, India, Mar. 2021, pp. 1661–1664. doi: 10.1109/ICACCS51430.2021.9441840.
[42] J. Nwokeji, F. Aqlan, A. Anugu, and A. Olagunju, ‘Big Data ETL Implementation Approaches: A Systematic Literature Review (P)’, presented at the The 30th International Conference on Software Engineering and Knowledge Engineering, Jul. 2018, pp. 714–721. doi: 10.18293/SEKE2018-152.
[43] X. Li and Y. Mao, ‘Real-Time data ETL framework for big real-time data analysis’, in *2015 IEEE International Conference on Information and Automation*, Lijiang, China, Aug. 2015, pp. 1289–1294. doi: 10.1109/ICInfA.2015.7279485.
[44] J. Wang and B. Liu, ‘Design of ETL Tool for Structured Data Based on Data Warehouse’, in *Proceedings of the 4th International Conference on Computer Science and Application Engineering*, Sanya China, Oct. 2020, pp. 1–5. doi: 10.1145/3424978.3425101.
[45] S. K. Bansal and S. Kagemann, ‘Integrating Big Data: A Semantic Extract-Transform-Load Framework’, *Computer*, vol. 48, no. 3, pp. 42–50, Mar. 2015, doi: 10.1109/MC.2015.76.
[46] E. F. Codd, ‘A Relational Model of Data for Large Shared Data Banks’, vol. 13, no. 6, p. 11, 1970.
[47] W. Ali, M. U. Shafique, M. A. Majeed, and A. Raza, ‘Comparison between SQL and NoSQL Databases and Their Relationship with Big Data Analytics’, *Asian J. Res. Comput. Sci.*, pp. 1–10, Oct. 2019, doi: 10.9734/ajrcos/2019/v4i230108.
[48] A. T. Alaklabi, ‘A Comparative Study of NoSQL databases’, *Biosci. Biotechnol. Res. Commun.*, vol. 12, no. 1, pp. 17–26, Feb. 2019, doi: 10.21786/bbrc/12.1/7.
[49] H. Fatima and K. Wasnik, ‘Comparison of SQL, NoSQL and NewSQL databases for internet of things’, in *2016 IEEE Bombay Section Symposium (IBSS)*, Baramati, India, Dec. 2016, pp. 1–6. doi: 10.1109/IBSS.2016.7940198.
[50] N. Leavitt, ‘Will NoSQL Databases Live Up to Their Promise?’, *Computer*, vol. 43, no. 2, pp. 12–14, Feb. 2010, doi: 10.1109/MC.2010.58.
[51] R. P. Padhy, M. R. Patra, and S. C. Satapathy, ‘RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database’s’, *Vol No*, no. 11.
[52] K. Kaur and D. M. Sachdeva, ‘Performance Evaluation of NewSQL Databases’, p. 5.
[53] A. Pavlo and M. Aslett, ‘What’s Really New with NewSQL?’, *ACM SIGMOD Rec.*, vol. 45, no. 2, pp. 45–55, Sep. 2016, doi: 10.1145/3003665.3003674.

[54] A. Varangaonkar, ‘NewSQL: What the hype is all about’, *Packt Hub*, Nov. 06, 2017. https://hub.packtpub.com/newsql-what-hype-about/ (accessed Dec. 21, 2022).
[55] Y. Li and S. Manoharan, ‘A performance comparison of SQL and NoSQL databases’, in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, Victoria, BC, Canada, Aug. 2013, pp. 15–19. doi: 10.1109/PACRIM.2013.6625441.
[56] S. Rautmare and D. D. M. Bhalerao, ‘MySQL and NoSQL database comparison for IoT application’, p. 4, 2016.
[57] M. Z. Li and M. S. Ryerson, ‘Reviewing the DATAS of aviation research data: Diversity, availability, tractability, applicability, and sources’, *J. Air Transp. Manag.*, vol. 75, no. November 2018, pp. 111–130, 2019, doi: 10.1016/j.jairtraman.2018.12.004.
[58] M. Ben Abda, P. P. Belobaba, and W. S. Swelbar, ‘Impacts of LCC growth on domestic traffic and fares at largest US airports’, *J. Air Transp. Manag.*, vol. 18, no. 1, pp. 21–25, Jan. 2012, doi: 10.1016/j.jairtraman.2011.07.001.
[59] Q. Fu and A. M. Kim, ‘Supply-and-demand models for exploring relationships between smaller airports and neighboring hub airports in the U.S.’, *J. Air Transp. Manag.*, vol. 52, pp. 67–79, Apr. 2016, doi: 10.1016/j.jairtraman.2015.12.008.
[60] L. Cadarso, V. Vaze, C. Barnhart, and Á. Marín, ‘Integrated Airline Scheduling: Considering Competition Effects and the Entry of the High Speed Rail’, *Transp. Sci.*, vol. 51, no. 1, pp. 132–154, Feb. 2017, doi: 10.1287/trsc.2015.0617.
[61] J. Rakas, A. Bauranov, and B. Messika, ‘Failures of critical systems at airports: Impact on aircraft operations and safety’, *Saf. Sci.*, vol. 110, pp. 141–157, Dec. 2018, doi: 10.1016/j.ssci.2018.05.022.
[62] M. M. Eshow, N. Ames, and M. Field, ‘ARCHITECTURE AND CAPABILITIES OF A DATA WAREHOUSE FOR ATM Sherlock Data Sources’, pp. 1–14, 2014.
[63] R. T. Thota, G. Bawa, and R. S. Stansbury, ‘Design and Prototyping of an Aviation Big Data Repository’, in *AIAA Scitech 2020 Forum*, Orlando, FL, Jan. 2020. doi: 10.2514/6.2020-0319.
[64] K. D. Kuhn, ‘A methodology for identifying similar days in air traffic flow management initiative planning’, *Transp. Res. Part C Emerg. Technol.*, vol. 69, pp. 1–15, Aug. 2016, doi: 10.1016/j.trc.2016.05.014.
[65] Y. Pang, H. Yao, J. Hu, and Y. Liu, ‘A Recurrent Neural Network Approach for Aircraft Trajectory Prediction with Weather Features From Sherlock’, in *AIAA Aviation 2019 Forum*, Dallas, Texas, Jun. 2019. doi: 10.2514/6.2019-3413.
[66] A. D. Evans and P. U. Lee, ‘Analyzing Double Delays at Newark Liberty International Airport’, in *16th AIAA Aviation Technology, Integration, and Operations Conference*, Washington, D.C., Jun. 2016. doi: 10.2514/6.2016-3456.
[67] H. M. (ARC-A. Arneson, ‘Sherlock Data Warehouse-2018’, Jun. 2018.

[68] H. M. (ARC-A. Arneson, ‘Sherlock Data Warehouse Overview-2019’, Apr. 2019. [Online]. Available: https://ntrs.nasa.gov/api/citations/20190025090/downloads/20190025090.pdf

[69] T. Larsen, ‘Cross-platform aviation analytics using big-data methods’, in *2013 Integrated Communications, Navigation and Surveillance Conference (ICNS)*, Herndon, VA, Apr. 2013, pp. 1–9. doi: 10.1109/ICNSurv.2013.6548579.

[70] A. Tyagi and J. Nanda, ‘ATLAS: Big Data Storage and Analytics Tool for ATM Researchers’, in *AIAA Infotech @ Aerospace*, San Diego, California, USA, Jan. 2016. doi: 10.2514/6.2016-0577.

[71] S. Ayhan, J. Pesce, P. Comitz, D. Sweet, S. Bliesner, and G. Gerberick, ‘Predictive analytics with aviation big data’, *Integr. Commun. Navig. Surveill. Conf. ICNS*, 2013, doi: 10.1109/ICNSurv.2013.6548556.

[72] U.S. Department of Commerce: National Oceanic and Atmospheric, Ed., ‘New Priorities for the 21 st Century: NOAA’s Strategic Plan’, Sep. 2004.

[73] M. Keel, G. Gimmestad, E. Stancil, A. Eckert, and M. Brown, ‘Aviation Weather Information Requirements Study’, no. June, pp. 1–178, 2000.

[74] W. Frost and D. W. Camp, ‘Proceedings: Sixth Annual Workshop on Meteorological and Environmental Inputs to Aviation Systems’, p. 151, 1983.

[75] ‘NOAA’s Weather and Climate Toolkit (Viewer and Data Exporter)’. https://www.ncdc.noaa.gov/wct/index.php (accessed Jun. 29, 2022).

[76] ‘Aircraft Situation Display to Industry: Functional Description and Interface Control Document’, Volpe Center Automation Application Division, ASDI-FD-001, 2000.

[77] ‘Enhanced Traffic Management System (ETMS) : functional description’. Jun. 30, 1995.

[78] ‘14 CFR 91.225 -- Automatic Dependent Surveillance-Broadcast (ADS-B) Out equipment and use.’ https://www.ecfr.gov/current/title-14/chapter-I/subchapter-F/part-91/subpart-C/section-91.225 (accessed Jun. 13, 2022).

[79] ‘14 CFR 91.227 -- Automatic Dependent Surveillance-Broadcast (ADS-B) Out equipment performance requirements.’ https://www.ecfr.gov/current/title-14/chapter-I/subchapter-F/part-91/subpart-C/section-91.227 (accessed Jun. 13, 2022).

[80] Federal Aviation Administration, ‘Automatic Dependent Surveillance-Broadcast(ADS-B) Flight Inspection’. Oct. 19, 2014.

[81] ‘ASDI Active Subscribers and Contracts’. Federal Aviation Administration, Sep. 02, 2012. [Online]. Available: https://www.fly.faa.gov/ASDI/asdidocs/ASDI_Active_Subscribers_and_Contacts.pdf

[82] ‘The OpenSky Network’. https://opensky-network.org/about/about-us (accessed Oct. 26, 2021).

[83] ‘Accessing Data Collected by ADS-B Exchange’, *ADS-B Exchange*. https://www.adsbexchange.com/data/ (accessed Jun. 14, 2022).

[84] ‘ADS-B Flight Tracking’, *FlightAware*. http://flightaware.com/adsb/ (accessed Jun. 14, 2022).

[85] ‘Structured Data vs. Unstructured Data: what are they and why care?’, *lawtomated*, Apr. 07, 2019. https://lawtomated.com/structured-data-vs-unstructured-data-what-are-they-and-why-care/ (accessed Jun. 13, 2022).

[86] ‘NEXRAD Data Inventory Search | National Centers for Environmental Information’. https://www.ncdc.noaa.gov/nexradinv/ (accessed Jul. 04, 2022).

[87] ‘What is Amazon S3? - Amazon Simple Storage Service’. https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#CoreConcepts (accessed Sep. 28, 2021).

[88] ‘Global Infrastructure Regions & AZs’, *Amazon Web Services, Inc.* https://aws.amazon.com/about-aws/global-infrastructure/regions_az/ (accessed Jul. 04, 2022).

[89] ‘AWS Command Line Interface’, *Amazon Web Services, Inc.* https://aws.amazon.com/cli/ (accessed Jul. 04, 2022).

[90] ‘Impala - Overview’. https://www.tutorialspoint.com/impala/impala_overview.htm (accessed Jul. 04, 2022).

[91] *impala-shell: Impala Shell*. Accessed: Jul. 04, 2022. [MacOS :: MacOS X, POSIX :: Linux]. Available: https://impala.apache.org/

[92] ‘Data elements and interchange formats - Information interchange - Representation of dates and times - Part2:Extensions’, Oct. 20, 2017. https://web.archive.org/web/20171020000043/https://www.loc.gov/standards/datetime/ISO_DIS%208601-2.pdf (accessed Jul. 19, 2022).

[93] ‘General Concept of POSIX time’. https://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xbd_chap04.html (accessed Jul. 19, 2022).

[94] ‘Maps, geocoding, and navigation APIs & SDKs | Mapbox’. https://www.mapbox.com/ (accessed Oct. 29, 2022).

[95] N. US Department of Commerce, ‘Past Significant Weather Events’. https://www.weather.gov/mob/events (accessed Dec. 22, 2022).

[96] ‘SpatiaLite: SpatiaLite’. https://www.gaia-gis.it/fossil/libspatialite/index (accessed Oct. 29, 2022).

[97] ‘Welcome to the Matplotlib Basemap Toolkit documentation — Basemap Matplotlib Toolkit 1.2.1 documentation’. https://matplotlib.org/basemap/ (accessed Oct. 29, 2022).

[98] T. wxPython Team, ‘Welcome to wxPython!’, *wxPython*, Aug. 02, 2021. https://wxpython.org/index.html (accessed Oct. 29, 2022).

# Appendix

# Appendix A: NEXRAD data extraction

```
import os

import subprocess


# #define function to use bash command to run AWS CLI to download raw weather radar data

def awc_auto_download(base_dir, dir_name):

    # base_dir= directory to store files download from aws

    print("created new directory for storting data: /WeatherRadarData/"+dir_name)

    if not os.path.exists(base_dir+dir_name):

        os.makedirs(base_dir+dir_name)

    # #change working directory to the created directroy

    os.chdir(base_dir + dir_name)

    print("current working directory: ", base_dir+dir_name)

    subprocess.run("aws s3 cp s3://noaa-nexrad-level2/"+dir_name+"/ . --recursive --no-sign-request", shell=True)

    # #reset working directory to the base directory

    os.chdir(base_dir)

    reset_dir=os.getcwd()

    print("Download process completed, reset to base directory: ", reset_dir)


if __name__ == '__main__':

    # #base_dir = directory to store files download from aws

    base_dir = "C:/Users/Sim/Documents/Code/Research/DataExtraction/WeatherRadarData/"

    # #select download date

    year = "2020"

    month = "10"

    date = "02"

    # #enter specific weather station, or "ALL" to download data of all weather station

    # #e.g.: station = "ALL" or station =["TEWR","TOKX","TJFK","KDIX","TPHL"]

    station = ['KOKX']

    print('-----------------------------------------------------------------')

    if station == "ALL":

        dir_name = year + "/" + month + "/" + date

        print('downloading weather radar data from all stations')
```

```
        print("prepare to download weather radar date of: " + dir_name)
        awc_auto_download(base_dir, dir_name)
else:
        print('downloading weather radar data from '+str(len(station))+' stations')
        for i in range (len(station)):
                curr_station = station[i]
                dir_name = year + "/" + month + "/" + date + "/" + curr_station
                print("prepare to download weather radar date of: " + dir_name)
                awc_auto_download(base_dir, dir_name)
```

# Appendix B: ADS-B data extraction

## Appendix B1: ADS-B data extraction with Python wrapper

```
# #Link to the OpenSky Network wrapper:   https://github.com/junzi40,-75s/pyopensky
# !!!!!!!!! Read the documentation on the page !!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# !!!! The wrapper required to manually modify a config file after installation !!!!!
# ####################################################################################
from pyopensky import OpenskyImpalaWrapper
from datetime import datetime
# #specifying date and time of interest
start_time = "2018-07-01 13:00:00"
end_time = "2018-07-01 13:01:00"
max_lat = 42
min_lat = 40
max_lon = -72
min_lon = -75

if __name__ == '__main__':
    # #initialzie OpenSky impala wrapper
    opensky = OpenskyImpalaWrapper()
    # #execute query and store query result to dataframe
    adsb_df = opensky.query(
        type="adsb",
        start=start_time,
        end=end_time,
        bong = [min_lat,min_lon,max_lat,max_lon]
    )
    # #generate file name by using start time
    # #file name format: absb + UnixTimestampOfStartTime.csv
    start_time_unix = datetime.fromisoformat(start_time).timestamp()
    file_name = 'adsb_'+str(start_time_unix)+'.csv'
    # #output csv file to computer
    adsb_df.to_csv(file_name, index=False)
```

# Appendix B2: ADS-B data extraction with custom Python script

```
import paramiko
from datetime import datetime
import pandas as pd
# #### user input for querying data from database #####################
# #1. time: YYYY-MM-DD HH:MM:SS+TimeZoneOffset
cmd_start_time = "2020-02-01 13:00:00+00:00"
cmd_end_time = "2020-02-01 13:01:00+00:00"
# #convert time to unix timestamp
cmd_start_time_unix = datetime.fromisoformat(cmd_start_time).timestamp()
cmd_end_time_unix = datetime.fromisoformat(cmd_end_time).timestamp()
# #2. geographical location
cmd_lat_min = "40"
cmd_lat_max = "42"
cmd_lon_min = "-75"
cmd_lon_max = "-72"
# #3. OpenSky Network login credentials
username = "AC-0636"
password = 'adsbsimulation'
# username = "username"
# password = "password"
# #4. path to storage directory
base_dir = 'C:/Users/Sim/Documents/Code/Research/DataExtraction/AdsbData/'
file_name = 'adsb' + str(cmd_start_time_unix) + '.csv'

if __name__ == '__main__':
    # #step1: connect to the opensky database server
    # #1.1 define parameters request for server connection
    # #Server address and port to opensky network can be found in: https://opensky-network.org/data/impala
    host = "data.opensky-network.org"
    port = 2230
    # # 1.2 initialize a ssh connection
    ssh = paramiko.SSHClient()
```

```
# # 1.3 set connection policy when connect to a unknown server
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
# #1.4 connect to host through a specific port with username and password
ssh.connect(host,port,username,password)
print('connected to OpenSky sever')

# ########## step 2: query data from database ###########
# #2.1 build the query
cmd = "-q SELECT time, icao24, lat, lon, velocity, heading, vertrate, callsign, baroaltitude, geoaltitude, onground, lastposupdate, hour" \
    +" FROM state_vectors_data4 " \
    + " WHERE lat > " + cmd_lat_min + " AND lat < " + cmd_lat_max \
    + " AND lon > " + cmd_lon_min + " AND lon < " + cmd_lon_max   \
    + " AND time > " + str(cmd_start_time_unix) + " AND time < " + str(cmd_end_time_unix)\
    + " AND geoaltitude IS NOT NULL AND velocity IS NOT NULL and vertrate IS NOT NULL AND onground = false;"
print('start to query data....')
# #2.2 execute command using ssh.exec_command
stdin, stdout, stderr = ssh.exec_command(cmd)
# #2.3 turn query output(stdout) to output
out = stdout.read().decode()
print("query done")
# #2.4 split query result using ' ', and turn the result to a list
# #because the result return from query are plain text with space inbetween columns
splited_out=out.split()

# #2.5 clear table edge in output
temp_out=[]
border_edge = splited_out[0]
mid_border_edge = "+--"
border_mid = '|'

for i in range(len(splited_out)):
    if splited_out[i] not in (border_edge,border_mid):
        if splited_out[i][:3] != mid_border_edge:
            temp_out.append(splited_out[i])
```

```
# #2.6 turn the clean list to a 'table' format
cleaned_out = []
# #step size 13 because there are 13 data column in the current dataset
for i in range (0, len(temp_out),13):
    cleaned_out.append(temp_out[i:i+13])
print(len(cleaned_out), " data record query from OpenSky Network database")

# #devide data into tables with 1025 rows because this is how the data is organized in the OpenSky Network
table_num = len(cleaned_out)//1025

# #remove the header row of each table
# #pop_index = the list of row index of the header rows
pop_index=[]
for i in range(1, table_num+1):
    pop_index.append(i*1025)
# print("index to be removed: ", pop_index)
cleaned_out.pop(pop_index)

# #use shift to update pop index as the list change while poping elements
shift = 0
for i in pop_index:
    cleaned_out.pop(i-shift)
    shift += 1

# #2.7 output clean data to csv file
print("generating csv file")
# #2.7.1 convert the data list to a dataframe
df = pd.DataFrame(cleaned_out[1:])
# #use the first row of the table to set the column name
df.columns = cleaned_out[0]

# #convert dataframe to csv file
df.to_csv(base_dir+file_name, index=False)
```

# Appendix C: Weather radar data processing

```
import os
import subprocess
import pandas as pd
from datetime import datetime
import dateutil.parser as dp

# #base_dir = directory to store files download from aws
base_dir = "C:/Users/Sim/Documents/Code/Research/DataExtraction/WeatherRadarData/"
# #wct_dir = directory of the wct weather toolkit
wct_dir = "C:/Users/Sim/Documents/Code/Research/wct-4.6.0"

# #define function to use bash command to run WCT to decode raw weather radar data
# #export file depends on the config set to wctBatchConfig-allPoints.xml under the wct-4.6.0 file
def wct_auto_convert(base_dir, date_dir, wct_dir):
    # #input directory of a specific date -> dir_name = YYYY/MM/DD
    input_dir = base_dir + date_dir

    # #append the name of weather staion directory under input_dir to a list
    ws_list = []
    for ws_dir in os.listdir(input_dir):
        ws_list.append(ws_dir)

    # #create new directory with the weather station (if not exist) for storing converted csv files
    csvoutput_dir = base_dir + 'output_csv/' + date_dir
    if not os.path.exists(csvoutput_dir):
        os.makedirs(csvoutput_dir)
    for i in range (len(ws_list)):
        ws_dir=csvoutput_dir+'/'+ws_list[i]
        if not os.path.exists(ws_dir):
            os.mkdir(ws_dir)

    # #use wct export bash command to convert nexradii files to csv files
```

```
    # #1. for each weather station directory
    for i in range (len(ws_list)):
        print('Decrypting file under |'+ws_list[i]+"| directory")
        # #2. update input and output path to ws directory
        station_input_dir = input_dir + '/' + ws_list[i]
        station_output_dir = csvoutput_dir+'/'+ ws_list[i]
        # #3. use wct bash command to convert files
        for nexradii_file in os.listdir(station_input_dir):
            # #3-1. get input file path
            file_path = station_input_dir + '/' + nexradii_file
            # #3-2. constract wct bash command and run command using subprocess
            # #nexradii_file[:-3] -> keep original file name, without '.gz'
            wct_command = "wct-export.bat "+ file_path + " " + station_output_dir+'/'+nexradii_file + " csv " +
wct_dir+'/wctBatchConfig-allPoints.xml'
            subprocess.run(wct_command, shell=True, cwd=wct_dir)
            # #delete .prj file (a not needed file generated during .csv export process)
            if os.path.exists(station_output_dir+'/'+nexradii_file+'.prj'):
                os.remove(station_output_dir+'/'+nexradii_file+'.prj')
def nexrad_standardized(base_dir,date_dir,station):
    # #path to de decoded weather radar file
    nexrad_dir = base_dir + 'output_csv/' + date_dir + '/' + station + '/'
    print(nexrad_dir)

    # #get all the nexradii file name under nexrad_dir directory
    file_list = []
    for nexrad_file in os.listdir(nexrad_dir):
        file_list.append(nexrad_file)

    for i in range(len(file_list)):
        nexrad_df = pd.read_csv(nexrad_dir+file_list[i])

        # #switch column order to avoid error loading longitude to map(if using csv file)
        nexrad_df=nexrad_df[['sweep', 'sweepTime', 'elevAngle', 'value', 'radialAng', 'surfaceRan', 'heightASL', 'latitude',
'longitude','heightRel']]
        # #drop undesired data column
```

```
            nexrad_df.drop(columns=['sweep','elevAngle','radialAng','surfaceRan'], inplace=True)
            # #rename sweepTime to Time
            nexrad_df.rename(columns={'sweepTime':'Time', 'value':'Reflectivity', 'latitude': 'Latitude', 'longitude': 'Longitude', 'heightASL':'HeightASL'}, inplace=True)
            # #time in one nexrad file are the same for all row
            time = nexrad_df.loc[0,'Time']
            # #convert time to unix timestamp
            time_unix = datetime.fromisoformat(str(dp.parse(time))).timestamp()
            # #change time to unix timestamp in dataframe
            nexrad_df['Time'] = nexrad_df['Time'].replace({time:time_unix})
            nexrad_df.to_csv(nexrad_dir+file_list[i],index=False)


if __name__ == '__main__':
    # #select date and station
    year = "2020"
    month = "10"
    date = "01"
    date_dir = year+"/"+month+"/"+date
    station = 'KOKX'

    # #run WCT by calling wct_auto_convert function
    print('Running WCT to decrypt raw weather data...')
    wct_auto_convert(base_dir=base_dir,date_dir=date_dir,wct_dir=wct_dir)

    # #process the decoded weather data
    print('Processing decoded weather radar file...')
    nexrad_standardized(base_dir=base_dir,date_dir=date_dir,station=station)
```

# Appendix D: Loading data to the database for storage

```
from os.path import exists
from datetime import datetime
import sqlite3
import pandas as pd
pd.set_option('float_format','{:.2f}'.format)
pd.set_option('max_columns',None)

# #define the name of the database
db_name = 'adsb_weather_db.sqlite'
# #define data capture time, need to use this to make the 'Unix_time' table
start_time = "2020-08-01 00:00:00+00:00"
end_time = "2020-08-05 00:00:00+00:00"

# #define the geographical region, need to use this to make the 'Geographical_point' table
min_lat = 37
max_lat = 44
min_lon = -79
max_lon = -70

# # path to the adsbdata file
adsb_base_dir ="C:/Users/Sim/Documents/Code/Research/DataExtraction/AdsbData/"
adsb_csv = "adsb_1596294000.csv"

# #path to the weather data file
nexrad_base_dir = "C:/Users/Sim/Documents/Code/Research/DataExtraction/WeatherRadarData/output_csv/"
nexrad_csv = "KDIX_2020-08-01_000432.csv"

# #create the database if not exist
def createDB():
    print('Database file NOT FOUND. Creating new database....\n')
    conn = sqlite3.connect(db_name)
    c = conn.cursor()
    # #enable load extension
    conn.enable_load_extension(True)
    # #load spatialite extension and initialize spatialite
    c.execute('SELECT load_extension("mod_spatialite")')
    conn.execute('SELECT InitSpatialMetaData(1)')
    # #create table
    c.execute('CREATE TABLE IF NOT EXISTS Unix_time(Time_ID text, Hour int)')
```

```
    c.execute('CREATE TABLE IF NOT EXISTS Geographical_point(Geo_ID text, Latitude int, Longitude int)')
    c.execute('CREATE TABLE IF NOT EXISTS Standardized_ADSB_data(Geo_ID text, Time_ID text, Time int, Latitude float,Longitude float, onground bool, Icao24 text, Velocity float, Heading float, Vertrate float, Callsign float, BarometricAltitude float, GeometricAltitude float, LastPosUpdate int)')
    c.execute('CREATE TABLE IF NOT EXISTS Standardized_NEXRADII_data(Geo_ID text, Time_ID text, Time int, Latitude float, Longitude float, HeightASL float, Reflectivity float)')
    conn.close()
    print('Done initializing new database, prepare to insert data...\n')
    return

# #check the number of time record in database
# #for identifying the last index number in table to make unique time_id
def checkTimeIndex(db):
    conn = sqlite3.connect(db)
    c = conn.cursor()
    query = c.execute('SELECT COUNT(*) FROM Unix_time')
    time_id = query.fetchone()[0]
    conn.close()
    return time_id

# #check the number of geo record in database
# #for identifying the last index number in table to make unique geo_id
def checkGeoIndex(db):
    conn = sqlite3.connect(db)
    c = conn.cursor()
    query = c.execute('SELECT COUNT(*) FROM Geographical_point')
    geo_id = query.fetchone()[0]
    conn.close()
    return geo_id

# #covert the select time period to hours and then unix timestamp
def generateHourList(start_t, end_t):
    start_time_unix = datetime.fromisoformat(start_t).timestamp()
    end_time_unix = datetime.fromisoformat(end_t).timestamp()
    # #calculate the number of hours in the input time
    hours = (end_time_unix-start_time_unix)/3600
    # time_df = pd.DataFrame(columns=['Hour'])
    hour_list =[]
    for i in range(int(hours)):
        temp_hour = start_time_unix+i*3600
        hour_list.append(temp_hour)
```

```
            # time_df.loc[-1] = [temp_hour]
            # time_df.index+=1
        return hour_list

# #create geo_list by matching all the combination of the select latitude and longitude 1 degree by 1 degree
def generateGeoList(min_lat,max_lat,min_lon,max_lon):
        geo_list =[]
        for lat in range (min_lat, max_lat+1):
            for lon in range (min_lon, max_lon+1):
                geo_list.append([lat,lon])
        return geo_list

# # check if the selected time period already exist in 'Unix_time' table
def checkTimeExistence(db, hour_list):
        conn = sqlite3.connect(db)
        c = conn.cursor()
        pop_index=[]
        for i in range (len(hour_list)):
            query = "SELECT Time_ID FROM Unix_Time WHERE Hour=" + str(hour_list[i]) + ';'
            exec_query = c.execute(query)
            # print(type(exec_query.fetchone()))
            result = exec_query.fetchone()
            if result is not None:
                # print('data record exit, skip this')
                pop_index.append(i)
            # elif result is None:
            #       print('no data record, keep this')
        # #pop data that already exist in the database
        if len(pop_index)>0:
            shift =0
            for i in pop_index:
                hour_list.pop(i - shift)
                shift+=1
        return hour_list

# # check if the selected geo area already exist in 'Unix_time' table
def checkGeoExistence(db, geo_list):
        conn = sqlite3.connect(db)
        c = conn.cursor()
        pop_index = []
        for i in range(len(geo_list)):
```

```
            query = "SELECT Geo_ID FROM Geographical_point WHERE Latitude=" + str(geo_list[i][0]) + ' AND Longitude=' \
                    + str(geo_list[i][1]) + ';'
            exec_query = c.execute(query)
            # print(type(exec_query.fetchone()))
            result = exec_query.fetchone()
            if result is not None:
                print('data record exit, skip this')
                pop_index.append(i)
            # elif result is None:
            #       print('no data record, keep this')
            if len(pop_index) > 0:
                shift = 0
                for i in pop_index:
                    geo_list.pop(i - shift)
                    shift += 1
        return geo_list

# #update Unix_time table in the database
def updateTimeTable(hour_list, time_id,db):
        id_list = []
        for i in range (len(hour_list)):
            id_list.append('T'+str(time_id))
            time_id+=1
        hour_df = pd.DataFrame()
        hour_df['Time_ID'] = id_list
        hour_df['Hour'] = hour_list
        conn = sqlite3.connect(db)
        hour_df.to_sql('Unix_time', conn, if_exists='append', index=False)
        conn.close()
        return

# # update 'geographical_point' table in the database
def updateGeoTable(geo_list,geo_id,db):
        id_list = []
        geo_df = pd.DataFrame(geo_list,columns=['Latitude','Longitude'])
        for i in range(len(geo_list)):
            id_list.append('G'+str(geo_id))
            geo_id+=1
        geo_df.insert(loc=0, column='Geo_ID',value=id_list)
        # print(geo_df)
        conn =sqlite3.connect(db)
```

```
        geo_df.to_sql('Geographical_point', conn, if_exists='append', index=False)
        conn.close
        return

# #insert adsb data to 'Standardized_ADSB_data' table
def insertAdsbData(adsb_file,db):
        # #set low_memory to False aviod mixing data type
        adsb_df = pd.read_csv(adsb_file,low_memory=False)
        # #create Time_ID and Geo_ID column in dataframe
        adsb_df['Time_ID'] = pd.NaT
        adsb_df['Geo_ID'] = pd.NaT
        # #rename column name in dataframe
adsb_df.rename(columns={'time':"Time","icao24":"Icao24","lat":"Latitude","lon":"Longitude","velocity":"Velocity","heading":"
Heading", "vertrate":"Vertrate", "callsing":"Callsign",
"baroaltitude":"BarometricAltitude","geoaltitude":"GeometricAltitude","lastposupdate":"LastPosUpdate","hour":"Hour"},inplace
=True)

        # #connect to database
        conn = sqlite3.connect(db)
        c = conn.cursor()
        for i in range(0,len(adsb_df)):
                # #get the matching Time_ID from Unix_Time table
                hour = adsb_df.loc[i,'Hour']
                query = c.execute("SELECT Time_ID From Unix_Time WHERE Hour = " + str(hour)+';')
                timeId = query.fetchone()[0]
                adsb_df.at[i,'Time_ID'] = timeId

                # #get the matching Geo_ID from Geographical_point table
                lat = int(float(adsb_df.loc[i,'Latitude']))
                lon = int(float(adsb_df.loc[i,'Longitude']))
                query = c.execute("SELECT Geo_ID From Geographical_point WHERE Latitude = " + str(lat)+ " AND Longitude =
"+str(lon) +';')
                geoID = query.fetchone()[0]
                adsb_df.at[i,'Geo_ID']= geoID

        # #drop the hour column in dataframe since it is replaced by the Time_ID
        adsb_df.drop('Hour', axis=1,inplace=True)
        # #load dataframe data to database
        adsb_df.to_sql('Standardized_ADSB_data',conn,if_exists='append',index=False)
        conn.close()
        return
```

```
# #create spatialite blob for air traffic data
def addAdsbBlob(db):
        conn = sqlite3.connect(db)
        conn.enable_load_extension(True)
        conn.execute('SELECT load_extension("mod_spatialite")')
        conn.execute("SELECT AddGeometryColumn('Standardized_ADSB_data', 'Geo_blob', 4326, 'POINTZ', 'XYZ')")
        c=conn.cursor()

        query =c.execute('SELECT COUNT(*) FROM Standardized_ADSB_data;')
        row_num = query.fetchone()[0]

        for i in range(1,row_num+1):
                query_geo_info = "SELECT Latitude, Longitude, BarometricAltitude FROM Standardized_ADSB_data WHERE rowid
="+ str(i)+';'
                query_result = (c.execute(query_geo_info)).fetchall()
                lat = query_result[0][0]
                lon = query_result[0][1]
                alt = query_result[0][2]

                query_update = "UPDATE Standardized_ADSB_data SET Geo_blob = GeomFromText('POINTZ(" + str(lon) + " " + str(
                        lat) + " " + str(alt) + ")',4326) WHERE rowid = " + str(i)
                c.execute(query_update)
                conn.commit()
        conn.close()
        return

# #insert adsb data to 'Standardized_NEXRADII_data' table
def insertNexradData(nexrad_file, db):
        # #set low_memory to False aviod mixing data type
        nexrad_df = pd.read_csv(nexrad_file,low_memory=False)
        # #create Time_ID and Geo_ID column in dataframe
        nexrad_df['Time_ID'] = pd.NaT
        nexrad_df['Geo_ID'] = pd.NaT
        # #get nexrad time from dataframe
        time = nexrad_df.loc[0,'Time']
        hour = (int(time)//3600)*3600

        # #connect to database
        conn =sqlite3.connect(db)
        c = conn.cursor()
```

```
    # #get time id from database
    query = c.execute("SELECT Time_ID From Unix_Time WHERE Hour = " + str(hour) + ';')
    timeId = query.fetchone()[0]
    # #assign time id to dataframe
    nexrad_df['Time_ID'] = timeId

    for i in range (0, len(nexrad_df)):
        # #get the matching Geo_ID from Geographical_point table
        lat = int(float(nexrad_df.loc[i,'Latitude']))
        lon = int(float(nexrad_df.loc[i,'Longitude']))
        query = c.execute("SELECT Geo_ID From Geographical_point WHERE Latitude = " + str(lat)+ " AND Longitude = 
"+str(lon) +';')
        geoID = query.fetchone()[0]
        nexrad_df.at[i,'Geo_ID']= geoID

    # #drop the heightRel column in dataframe as it is not needed in dataframe
    nexrad_df.drop('heightRel', axis=1,inplace=True)
    # #load dataframe data to database
    nexrad_df.to_sql('Standardized_NEXRADII_data',conn,if_exists='append',index=False)
    return

# #create spatialite blob for weather radar data
def addNexradBlob(db):
    conn = sqlite3.connect(db)
    conn.enable_load_extension(True)
    conn.execute('SELECT load_extension("mod_spatialite")')
    conn.execute("SELECT AddGeometryColumn('Standardized_NEXRADII_data', 'Geo_blob', 4326, 'POINTZ', 'XYZ')")
    c=conn.cursor()

    query = c.execute('SELECT COUNT(*) FROM Standardized_NEXRADII_data;')
    row_num = query.fetchone()[0]

    for i in range(1,row_num+1):
        query_geo_info = "SELECT Latitude, Longitude, HeightASL FROM Standardized_NEXRADII_data WHERE rowid ="
+ str(i) + ';'
        query_result = (c.execute(query_geo_info)).fetchall()
        lat = query_result[0][0]
        lon = query_result[0][1]
        alt = query_result[0][2]

        query_update = "UPDATE Standardized_NEXRADII_data SET Geo_blob = GeomFromText('POINTZ(" + str(lon) + "
```

```
" + str(
                    lat) + " " + str(alt) + ")',4326) WHERE rowid = " + str(i)
            c.execute(query_update)
            conn.commit()

        conn.close()
        return

if __name__ == '__main__':
        # # check if database already exist
        db_exists = exists(db_name)
        if db_exists == False:
                createDB()
                time_id = 0
                geo_id = 0
        else:
                print('database already exist')
                time_id = checkTimeIndex(db=db_name)
                geo_id = checkGeoIndex(db=db_name)

        # #1. check and update information in the 'Unix_time' table
        time_list = generateHourList(start_time, end_time)
        checkTimeExistence(db=db_name, hour_list=time_list)
        updateTimeTable(hour_list=time_list,time_id=time_id,db=db_name)

        # #2. check and update information in the 'Geographical_point' table
        geo_list = generateGeoList(min_lat=min_lat,max_lat=max_lat,min_lon=min_lon,max_lon=max_lon)
        checkGeoExistence(db=db_name, geo_list=geo_list)
        updateGeoTable(geo_list=geo_list,geo_id=geo_id,db=db_name)

        # #3. load adsb data files, and load to database
        adsb_file = adsb_base_dir+adsb_csv
        insertAdsbData(adsb_file=adsb_file,db=db_name)
        addAdsbBlob(db=db_name)

        # #4. load weather data files, and load to database
        nexrad_file = nexrad_base_dir + nexrad_csv
        insertNexradData(nexrad_file=nexrad_file, db = db_name)
        addNexradBl
```

# Appendix E: Prototype #1

```
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
from datetime import datetime

# #1. load the selected data file and filter out the information that are outside the region
# #the data in the csv file are converted to dataframe to make it easier for the future steps
adsb_df = pd.read_csv('adsb csv/adsb_1590969660.csv')
adsb_df = adsb_df.loc[adsb_df['lat']>=24.93]
adsb_df = adsb_df.loc[adsb_df['lat']<=43.96]
adsb_df = adsb_df.loc[adsb_df['lon']>=-88.81]
adsb_df = adsb_df.loc[adsb_df['lon']<=-67.26]
adsb_df = adsb_df.loc[adsb_df['baroaltitude']<=14000]

# #2. create three list to store longitude, latitude, and altitude information separately
def load_adsb_from_df(time):
    select_df = adsb_df.loc[adsb_df['time']==time]
    lon = select_df['lon'].tolist()
    lat = select_df['lat'].tolist()
    alt = select_df['baroaltitude'].tolist()
    return lon, lat,alt

# #3. load the three lists to map
def load_data_to_map(i):
    # #create a map background for the selected geographical region
    map = Basemap(projection='mill',
                  llcrnrlat=24.93, urcrnrlat=43.96,
                  llcrnrlon=-88.81, urcrnrlon=-67.26,
                  fix_aspect=False)
```

```
# #create text label for New York and Miami
nyc=[-74.006111,40.712778]
miami=[-80.208615,25.775163]
nyc_x,nyc_y= map(nyc[0],nyc[1])
miami_x,miami_y= map(miami[0],miami[1])

# #update 'time' whenever the animation being called
# #the initial time comes from the file name of the selected data file
time = 1590969660+i

# #get adsb data from dataframe
lon, lat, alt = load_adsb_from_df(time)
# #convert the longitude and latitude information to 2D (x,y) format for the plot
x,y = map(lon,lat)

# #initialize the 3D map
fig = plt.gcf()
ax = Axes3D(fig)
ax.set_zlim3d(0, 15000)

# #define the view of the 3d map
ax.azim = 315
ax.elev = 20
ax.add_collection3d(map.drawcoastlines())

# #plot dot by using the information in the x,y, alt list
p=ax.scatter3D(x, y, alt, c=alt, cmap='Paired')
ax.set_title('Visualization of archived air traffic data',fontsize=20)
ax.text(nyc_x,nyc_y,0,'New York', ha='left', fontsize=20, weight='bold')
ax.text(miami_x,miami_y,0,'Miami',ha='right',fontsize=20,weight='bold')
ax.set_zlabel('Altitude above ground level(ft)')
ax.annotate(str(datetime.utcfromtimestamp(time))+' UTC',
            xy=(0.5, 0), xytext=(0, 10),
            xycoords=('axes fraction', 'figure fraction'),
```

```
                textcoords='offset points',
                size=25, ha='center', va='bottom',weight='bold')

    plt.colorbar(mappable=p,shrink = 0.8,pad=0.03)
    i+=1

if __name__ == '__main__':
    # #update the plot every 3 seconds
    ani = FuncAnimation(plt.gcf(), load_data_to_map, interval=3000)
    plt.show()
```

# Appendix F: Prototype #2

## Appendix F1: Graphical user interface

```
import wx
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.figure import Figure
import display_map
import get_db_data
import get_navi_charts

# #define the main frame (the gui)
class MainFrame(wx.Frame):
    def __init__(self, parent,title):
        #size(x,y) = x width * y heigh
        super(MainFrame, self).__init__(parent, title = "Flight Smart", size = (1400,1000))
        # #set min and max frame size to lock window size
        self.SetMinSize((1400,1000))
        self.SetMaxSize((1400,1000))
        ##set panel
        self.panel = MainPanel(self)
        ##set menu bar
        menuBar = wx.MenuBar()
        fileMenu = wx.Menu()
        exitMenuItem = fileMenu.Append(wx.NewId(), "Exit","Exit the application")
        menuBar.Append(fileMenu, "&More Settings")
        self.Bind(wx.EVT_MENU, self.onExit, exitMenuItem)
        self.SetMenuBar(menuBar)

    def onExit(self, event):
        self.Close()

# #defined the content to display in the main panel
class MainPanel(wx.Panel):
    def __init__(self, parent):
        super(MainPanel,self).__init__(parent)
        ## config of the main map
        self.mainmapFigure = Figure(figsize=(10, 10), dpi=100)
        self.mainmapCanvas = FigureCanvas(self, -1, self.mainmapFigure)
```

```
        self.mainmapAx = self.mainmapFigure.add_axes([0, 0, 1, 1])

        ## config of the sub map
        self.submapFigure = Figure(figsize=(4, 2), dpi=100)
        self.submapCanvas = FigureCanvas(self,-1,self.submapFigure)
        self.submapAx = self.submapFigure.add_axes([0,0,1,1])

        ##config of the elevation profile
        self.eleproFigure = Figure(figsize=(3.5,2))
        self.eleproCanvas = FigureCanvas(self,-1,self.eleproFigure)
        self.eleproAx = self.eleproFigure.add_axes([0,0,1,1])
        self.panelLayout()
        self.mainTimer = wx.Timer(self)
        self.Bind(wx.EVT_TIMER, self.updateTimer)
        self.mainTimer.Start(1000)

    def panelLayout(self):
        # #ctrlBox = box sizer for control manual
        ctrlBox = wx.BoxSizer(wx.VERTICAL)

        # #mainmapBox = box sizer for map
        mainmapBox = wx.BoxSizer(wx.VERTICAL)
        # #ADD the map figure to mainmapBox
        mainmapBox.Add(self.mainmapCanvas, 0, wx.LEFT)

        # #submapBox = box sizer for submap
        submapBox = wx.BoxSizer(wx.VERTICAL)
        submapBox.Add(self.submapCanvas,0,wx.Center)

        # #eleproBox = box sizer for elevation profile
        eleproBox = wx.BoxSizer(wx.VERTICAL)
        eleproBox.Add(self.eleproCanvas,0,wx.Center)

        # #hBox = "final" box sizer to display map and control horizontally
        hBox = wx.BoxSizer(wx.HORIZONTAL)

        # #cltBox = box sizer title for check lists
        cltBox = wx.BoxSizer(wx.HORIZONTAL)
        # #clcBox = box sizer choice for check lists
        clcBox = wx.BoxSizer(wx.HORIZONTAL)
```

```
        # #box sizer for vfr buttons
        vfrbtnBox = wx.BoxSizer(wx.HORIZONTAL)
        # #box sizer for ifr buttons
        ifrbtnBox = wx.BoxSizer(wx.HORIZONTAL)

        # #btnBox2 = box sizer for buttons: "Swap view", "Filght Plan Map"
        btnBox2 = wx.BoxSizer(wx.HORIZONTAL)

        # # checklist box title
        self.displayLabel = wx.StaticText(self,label = "Display option")
        cltBox.Add(self.displayLabel, 0, wx.EXPAND)
        cltBox.AddSpacer(100)
        self.mapviewLabel = wx.StaticText(self, label = "Map view option")
        cltBox.Add(self.mapviewLabel, 0, wx.EXPAND)
        # #add check list title box sizer to control box sizer
        ctrlBox.Add(cltBox)

        # #display option list
        displayList = ["Flight Plan", "Air Traffic (ADS-B)", "Enroute Icing", "Wind", "Turbulence", "Show
Runways","Show Waypoints"]
        # #markers for display choice, default set as flight plan, so the first marker is 1
        self.displayMarker=[1,0,0,0,0,0,0]
        self.displayChoice = wx.CheckListBox(self, choices=displayList)
        self.displayChoice.Check(0, True)
        # #Bind display choice from the radio box
        self.displayChoice.Bind(wx.EVT_CHECKLISTBOX, self.mapdisplayEvent)
        clcBox.Add(self.displayChoice, 0, wx.EXPAND)
        clcBox.AddSpacer(20)

        # #map view option list
        mapList = ["Terrain Map View","Street Map View", "Satellite Map View"]
        self.mapviewChoice = wx.CheckListBox(self, choices=mapList)
        # # mark the deafult map view as checked
        self.mapviewChoice.Check(0,True)
        # # marker to "remember" the current view option
        self.viewMarker=0
        # # bind check box list event
        self.mapviewChoice.Bind(wx.EVT_CHECKLISTBOX, self.mapviewEvent)
        clcBox.Add(self.mapviewChoice,0,wx.EXPAND)
        # # add check list choice sizer to control box sizer
        ctrlBox.Add(clcBox)
```

```
        ctrlBox.AddSpacer(10)

        # #"Time" stuff
        self.currenttimeLabel=wx.StaticText(self, label = "Current Time: ")
        ctrlBox.Add(self.currenttimeLabel,0,wx.EXPAND)
        self.currentTime=get_db_data.getcurrenttime()
        self.currenttimeDisplay = wx.StaticText(self, label = self.currentTime)
        ctrlBox.Add(self.currenttimeDisplay)
        ctrlBox.AddSpacer(10)

        self.timeoffsetLabel = wx.StaticText(self,label = "Time Offest(from now)")
        ctrlBox.Add(self.timeoffsetLabel,0,wx.EXPAND)
        timeoffsetList=["30 Minute before 'now'", "1 Hour before 'now'","None(Realtime)","30 Minute after
'now'", "1 Hour after 'now'"]
        self.timeoffsetChoice = wx.CheckListBox(self,choices=timeoffsetList)
        # #mark "None(Realtime) as default term"
        self.timeoffsetChoice.Check(2,True)
        ctrlBox.Add(self.timeoffsetChoice,0,wx.EXPAND)
        ctrlBox.AddSpacer(10)

        # #"Altitude selection" title
        self.altitudeLabel = wx.StaticText(self,label="Altitude Selection")
        ctrlBox.Add(self.altitudeLabel,0,wx.EXPAND)
        self.altitudeInput = wx.SpinCtrl(self,initial=15000, min=1000,max=20000)
        self.last_altitude = 15000
        self.altitudeInput.Bind(wx.EVT_SPINCTRL,self.spinctrlEvent)
        ctrlBox.Add(self.altitudeInput,0,wx.EXPAND)

        # #"Update map at this altitude" button
        self.updatealtBtn = wx.Button(self,label="Update Map at this Altitude")
        # self.updatealtBtn.Bind(wx.EVT_BUTTON,self.updatealtEvent)
        ctrlBox.Add(self.updatealtBtn)
        ctrlBox.AddSpacer(20)

        # #Navigation chart stuff
        self.navigationchartLabel = wx.StaticText(self,label = "Navigation Chart")
        ctrlBox.Add(self.navigationchartLabel,0,wx.EXPAND)

        self.vfrLabel = wx.StaticText(self, label = "VFR")
        ctrlBox.Add(self.vfrLabel,0,wx.EXPAND)
```

```
self.vfrsectionalBtn = wx.Button(self,label = "Sectional")
vfrbtnBox.Add(self.vfrsectionalBtn)
self.vfrsectionalBtn.Bind(wx.EVT_BUTTON,self.vfrsecEvent)
vfrbtnBox.AddSpacer(10)

self.vfrterminalBtn = wx.Button(self,label = "Terminal Area")
vfrbtnBox.Add(self.vfrterminalBtn)
self.vfrterminalBtn.Bind(wx.EVT_BUTTON,self.vfrterEvent)
vfrbtnBox.AddSpacer(10)

self.vfrterminalpBtn = wx.Button(self,label = "Terminal Planning")
vfrbtnBox.Add(self.vfrterminalpBtn)
self.vfrterminalpBtn.Bind(wx.EVT_BUTTON,self.vfrterplanEvent)
ctrlBox.Add(vfrbtnBox)
ctrlBox.AddSpacer(20)

self.ifrLabel = wx.StaticText(self,label = "IFR")
ctrlBox.Add(self.ifrLabel)
self.ifrlowaltBtn = wx.Button(self, label = "Low Altitude")
ifrbtnBox.Add(self.ifrlowaltBtn)
self.ifrlowaltBtn.Bind(wx.EVT_BUTTON,self.ifrlowaltEvent)
ifrbtnBox.AddSpacer(10)

self.ifrhialtBtn = wx.Button(self,label = "Hight Altitude")
ifrbtnBox.Add(self.ifrhialtBtn)
self.ifrhialtBtn.Bind(wx.EVT_BUTTON,self.ifrhialtEvent)
ctrlBox.Add(ifrbtnBox)
ctrlBox.AddSpacer(10)

# add sub map to the ctrlBox Sizer
ctrlBox.Add(submapBox)
ctrlBox.AddSpacer(10)

# #elevation profile title
self.eleproLabel = wx.StaticText(self,label="Elevation Profile")
ctrlBox.Add(self.eleproLabel,0,wx.EXPAND)

# #add elevation profile to the ctrlBox Sizer
ctrlBox.Add(eleproBox)

# #add them to the "final" Box
```

```
        hBox.Add(mainmapBox)
        hBox.Add(ctrlBox)
        self.SetSizer(hBox)

        # #display_default main map
        display_map.zoomflightplanMap(self,self.mainmapFigure,self.mainmapCanvas,self.mainmapAx)
        # #display default sub map
        display_map.fullflightplanMap(self,self.submapFigure,self.submapCanvas,self.submapAx)
        # #display elevation map
        display_map.plotelevationProfile(self,self.eleproFigure,self.eleproCanvas,self.eleproAx)

# #map view event: for changing map style marker
    def mapviewEvent(self,event):
        viewIndex = self.mapviewChoice.GetCheckedItems()
        ## find the current view index( viewMarker) from the list, and unchecked it
        removeindex = viewIndex.index(self.viewMarker)
        self.mapviewChoice.Check(viewIndex[removeindex],False)
        if removeindex == 0:
            self.viewMarker = viewIndex[1]
        else:
            self.viewMarker = viewIndex[0]

        viewResult = self.mapviewChoice.GetString(self.viewMarker)
        display_map.mapView(self,self.mainmapFigure,self.mainmapCanvas,self.mainmapAx,viewResult)

# #map display event
    def mapdisplayEvent(self, event):
        displayResult = self.displayChoice.GetCheckedStrings()
        if "Flight Plan" in displayResult and self.displayMarker[0] == 0:
            display_map.flightplanMap(self, self.mainmapFigure, self.mainmapCanvas, self.mainmapAx)
            self.displayMarker[0]=1
        if "Air Traffic (ADS-B)" in displayResult and self.displayMarker[1] == 0:
            display_map.plotTraffic(self, self.mainmapFigure, self.mainmapCanvas, self.mainmapAx)
            self.displayMarker[1] =1
        if "Enroute Icing" in displayResult and self.displayMarker[2] ==0:
            display_map.plotIcing(self,self.mainmapFigure,self.mainmapCanvas, self.mainmapAx,15000)
            self.displayMarker[2] =1
        elif "Show Waypoints" in displayResult and self.displayMarker[6] == 0:
            display_map.plotWaypoint(self, self.mainmapFigure, self.mainmapCanvas, self.mainmapAx)

## time update
```

```
    def updateTimer(self, event):
        self.currentTime = get_db_data.getcurrenttime()
        self.currenttimeDisplay.SetLabelText(self.currentTime)

## force altitude slider increment
    def spinctrlEvent(self, event):
        ##calculate the delta altitude to figure out whether "UP" or "DOWN" is clicked
        delta_alt = self.altitudeInput.GetValue() - self.last_altitude
        ##"UP"
        if delta_alt >0:
            self.last_altitude += 1000 ##we want to set the change step as 1000
            self.altitudeInput.SetValue(self.last_altitude)
        ##"DOWN"
        else:
            self.last_altitude -= 1000    ##we want to set the change step as 1000
            self.altitudeInput.SetValue(self.last_altitude)

## display Sectional VFR chart
    def vfrsecEvent(self, event):
        get_navi_charts.getvfrSectional()

## display Terminal VFR chart
    def vfrterEvent(self,event):
        get_navi_charts.getvfrTerminal()

## display Terminal Planning VFR chart
    def vfrterplanEvent(self,event):
        get_navi_charts.getvfrterminalPlanning()

## display Low Altitude IFR chart
    def ifrlowaltEvent(self,event):
        get_navi_charts.getifrlowAltitude()

## display High Altitude IFR chart
    def ifrhialtEvent(self,event):
        get_navi_charts.getifrhiAltitude()

## update map with altitude
    # def updatealtEvent(self,event):

class TApplication(wx.App):
```

```
    def OnInit(self):
        self.frame = MainFrame(parent=None, title=None)
        self.frame.Show()
        return True
if __name__ == "__main__":
    Application = TApplication(0)
    Application.MainLoop()
```

# Appendix F2: Design the main map

```
from mpl_toolkits.basemap import Basemap
from our_opensky_api import OpenSkyApi
import get_db_data
import get_elevation_profile
import matplotlib.image as image
import asyncio

##coordinate of departure & destination
##KJFK
depart_lat = 40.6444
depart_lon = -73.7867
##KLGA
dest_lat = 40.7844
dest_lon = -73.8767

##global depart_lat, depart_lon, dest_lat, dest_lon
depart_icao = "KJFK"
dest_icao = "KLGA"
map_view_mode = "World_Terrain_Base"

##current location representation
## currently set in JFK
current_lon = depart_lon
current_lat = depart_lat

global llclat, urclat, llclon, urclon
llclat = min(depart_lat, dest_lat)    # min_lat
urclat = max(depart_lat, dest_lat)    # max lat
llclon = min(depart_lon, dest_lon)    # min lon
urclon = max(depart_lon, dest_lon)    # max lon

def mapView(self, _figure, _canvas, _axes, _view):
    _figure.set_canvas(_canvas)
    _axes.clear()
    global map_view_mode
    if _view == "Street Map View":
        # map_view_mode = 'World_Street_Map'
        map_view_mode = '/Canvas/World_Dark_Gray_Base'
```

```
    elif _view == "Terrain Map View":
        map_view_mode = 'World_Terrain_Base'
    elif _view == "Satellite Map View":
        map_view_mode = 'World_Imagery'
    # plotbackgroundMap(self,_figure,_canvas,_axes)
    zoomflightplanMap(self, _figure, _canvas, _axes)

## config for the maps
def mapVariable(_llcrnlat, _urcrnrlat, _llcrnrlon, _urcrnrlon, _axes):
    _map = Basemap(llcrnrlat=_llcrnlat, urcrnrlat=_urcrnrlat,
                   llcrnrlon=_llcrnrlon, urcrnrlon=_urcrnrlon,
                   epsg=4269, ax=_axes)
    # _map.arcgisimage(service=map_view_mode, xpixels=2000, verbose=True)
    return _map


## plot the background map on the given area
def plotbackgroundMap(self, _figure, _canvas, _axes):
    _figure.set_canvas(_canvas)
    _axes.clear()

    # global llclat, urclat, llclon, urclon
    _map = mapVariable(llclat - 0.05, urclat + 0.05, llclon - 0.05, urclon + 0.05, _axes)
    _map.arcgisimage(service=map_view_mode, xpixels=2000, verbose=True)

    depart_x, depart_y = _map(depart_lon, depart_lat)
    dest_x, dest_y = _map(dest_lon, dest_lat)

    _axes.scatter(depart_x, depart_y, marker='D', color='r', s=20)
    _axes.annotate(depart_icao, xy=(depart_x, depart_y), color='r')
    _axes.scatter(dest_x, dest_y, marker='D', color='r', s=20)
    _axes.annotate(dest_icao, xy=(dest_x, dest_y), color='r')

    _figure.canvas.draw()
    return _map

## zoom map background set up, clear map with flight plan plotted
## currentlly using JFK as our center
def zoomflightplanMap(self, _figure, _canvas, _axes):
    _figure.set_canvas(_canvas)
    _axes.clear()
```

```
        # global llclat, urclat, llclon, urclon
        _map = mapVariable(depart_lat - 0.05, depart_lat + 0.05, depart_lon - 0.05, depart_lon + 0.05, _axes)
        _map.arcgisimage(service=map_view_mode, xpixels=2000, verbose=True)

        ## plot jfk
        depart_x, depart_y = _map(depart_lon, depart_lat)
        _axes.scatter(depart_x, depart_y, marker='D', color='r', s=50)
        _axes.annotate(depart_icao, xy=(depart_x, depart_y), color='r')

        ## plot flight plan
        _fplat, _fplon, _fpalt = get_db_data.getflightPlan()
        fp_x, fp_y = _map(_fplon, _fplat)
        _axes.plot(fp_x, fp_y, linewidth=1.5, color='r')

        ## plot our "current" location
        current_x, current_y = _map(current_lon, current_lat)
        _axes.scatter(current_x, current_y, marker='o', color='y', s=30)
        _figure.canvas.draw()
        return _map

##full map background set up, clear map with flight plan plotted
def fullflightplanMap(self, _figure, _canvas, _axes):
        _figure.set_canvas(_canvas)
        _axes.clear()
        _map = plotbackgroundMap(self, _figure, _canvas, _axes)
        _fplat, _fplon, _fpalt = get_db_data.getflightPlan()
        fp_x, fp_y = _map(_fplon, _fplat)
        _axes.plot(fp_x, fp_y, linewidth=1.5, color='r')
        _figure.canvas.draw()
        return _map


## elevation map set up
def plotelevationProfile(self, _figure, _canvas, _axes):
        _figure.set_canvas(_canvas)
        _axes.clear()
        _d_list_rev, _elev_list = get_elevation_profile.elevationData()

        # BASIC STAT INFORMATION
        mean_elev = round((sum(_elev_list) / len(_elev_list)), 3)
```

```
        min_elev = min(_elev_list)
        max_elev = max(_elev_list)
        distance = _d_list_rev[-1]

        _axes.plot(_d_list_rev, _elev_list)
        _axes.plot([0, distance], [min_elev, min_elev], '--g', label='min: ' + str(min_elev) + ' m')
        _axes.plot([0, distance], [max_elev, max_elev], '--r', label='max: ' + str(max_elev) + ' m')
        _axes.plot([0, distance], [mean_elev, mean_elev], '--y', label='ave: ' + str(mean_elev) + ' m')
        _axes.fill_between(_d_list_rev, _elev_list, 0, alpha=0.1)
        _axes.text(_d_list_rev[0], _elev_list[0], "KJFK")    ##P1
        _axes.text(_d_list_rev[-1], _elev_list[-1], "KLGA")    ##P2
        _axes.grid()
        _axes.legend(fontsize='small')
        _figure.canvas.draw()

##map ADS-B info
def plotTraffic(self, _figure, _canvas, _axes):
        _figure.set_canvas(_canvas)
        _axes.clear()

        ##plotting the icon to the graph
        dpi = 72;
        imageSize = (16, 16)
        wpt_im = image.imread('icon/plane.png')
        _map = fullflightplanMap(self, _figure, _canvas, _axes)
        lon = []
        lat = []
        tailnum = []
        alt = []
        j = 0
        api = OpenSkyApi()
        global llclat, urclat, llclon, urclon
        ##get live ADS-B from opensky api
        states = api.get_states(bbox=(llclat - 0.08, urclat + 0.08, llclon - 0.08, urclon + 0.08))
        for s in states.states:
                lon.append([])
                lon[j] = s.longitude
                lat.append([])
                lat[j] = s.latitude
                alt.append([])
                alt[j] = s.geo_altitude
```

```
            tailnum.append([])
            tailnum[j] = s.callsign
            j += 1

    ## map air traffic to the map
    traffic_x, traffic_y = _map(lon, lat)

    # plot icao code to the aircraft
    for k in range(len(traffic_x)):
        _axes.annotate(tailnum[k], xy=(traffic_x[k], traffic_y[k]), color='b')

    ## check for altitude value obtained from the api
    ## remove the 'None' value
    for l in range(0, j - 1):
        if alt[l] is None:
            alt[l] = 0
            l += 1
        else:
            l += 1

    ## plot the traffic in the map
    _axes.scatter(traffic_x, traffic_y, s=10, c=alt, cmap='Paired')

    ##Ref: https://stackoverflow.com/questions/2318288/how-to-use-custom-png-image-marker-with-plot
    points, = _axes.plot(traffic_x, traffic_y, "bo", mfc="None", mec="None", markersize=imageSize[0] * (dpi
/ 96))
    points._transform_path()
    path, affine = points._transformed_path.get_transformed_points_and_affine()
    path = affine.transform_path(path)
    for pixelPoint in path.vertices:
        _figure.figimage(wpt_im, pixelPoint[0] - imageSize[0] / 2, pixelPoint[1] - imageSize[1] / 2,
origin="upper")
    _figure.canvas.draw()

    return _map

## plot icing info
def plotIcing(self, _figure, _canvas, _axes, _alt):
    get_db_data.getIcing(llclat, llclon, urclat, urclon, _alt)

## plot waypoint info
```

```
def plotWaypoint(self, _figure, _canvas, _axes):
        _figure.set_canvas(_canvas)
        _axes.clear()

        ##plotting the icon to the graph
        dpi = 72;
        imageSize = (32, 32)
        wpt_im = image.imread('icon/waypoint.png')
        _map = fullflightplanMap(self, _figure, _canvas, _axes)
        _wpt_name_list, _wpt_lat_list, _wpt_lon_list = get_db_data.getWaypoint(llclat, llclon, urclat, urclon)

        ## map wpt to the map
        wpt_x, wpt_y = _map(_wpt_lon_list, _wpt_lat_list)

        # _axes.scatter(wpt_x, wpt_y, s=10, color = 'g')

        ## plot the traffic in the map
        points, = _axes.plot(wpt_x, wpt_y, "bo", mfc="None", mec="None", markersize=imageSize[0] * (dpi / 96))
        points._transform_path()
        path, affine = points._transformed_path.get_transformed_points_and_affine()
        path = affine.transform_path(path)
        for pixelPoint in path.vertices:
                _figure.figimage(wpt_im, pixelPoint[0] - imageSize[0] / 2, pixelPoint[1] - imageSize[1] / 2,
origin="upper")

        ## plot wpt name to the aircraft
        for k in range(len(wpt_x)):
                _axes.annotate(_wpt_name_list[k], xy=(wpt_x[k], wpt_y[k]), color='k')
        _figure.canvas.draw()

        return _map
```

# Appendix F3: Obtain airspace and weather data from data files

```
import pandas as pd
import numpy as np
import sqlite3
from datetime import datetime

## set cifp connection
cifp_path = "db/CIFP_v1.db"
cifp_conn = sqlite3.connect(cifp_path)

## set icing connection
icing_patn = "db/Icing2020-03-16.db"
icing_conn = sqlite3.connect(icing_patn)

## get the coordinate of the departure and destination
def getaptLocation(departicao, desticao):
    read_apt = pd.read_csv("airports.csv")
    ##"marker", use to know if done searching
    _getdepart = 0
    _getdest = 0
    ##use a list to save data
    location_list_lat_lon = []

    for i in range(0,len(read_apt)):
        if _getdepart == 0 or _getdest ==0:
            if read_apt["ident"][i] == departicao:
                departlat = read_apt["latitude_deg"][i]
                departlon = read_apt["longitude_deg"][i]
                _getdepart = 1
            elif read_apt["ident"][i] == desticao:
                destlat = read_apt["latitude_deg"][i]
                destlon = read_apt["longitude_deg"][i]
                _getdest = 1
            else:
                i+=1
        else: ##_getdepart = 1 and _getdest =1
            i = len(read_apt)

    if _getdepart ==1 and _getdest ==1:
```

```
            location_list_lat_lon.append([])
            location_list_lat_lon[0] = departicao
            location_list_lat_lon.append([])
            location_list_lat_lon[1] = departlat
            location_list_lat_lon.append([])
            location_list_lat_lon[2] = departlon
            location_list_lat_lon.append([])
            location_list_lat_lon[3] = desticao
            location_list_lat_lon.append([])
            location_list_lat_lon[4] = destlat
            location_list_lat_lon.append([])
            location_list_lat_lon[5] = destlon

    return location_list_lat_lon, _getdepart,_getdest
## get current time
def getcurrenttime():
    now = datetime.now()
    current_time=now.strftime("%c")
    return current_time

## get the given flight plan
def getflightPlan():
    fp = pd.read_csv("fp/KJFK-KLGA.csv")
    ##get data from flightplan.csv
    fplatarr = np.array(fp["latitude_deg"][0:200])
    fplonarr = np.array(fp["longitude_deg"][0:200])
    fpaltarr = np.array(fp["elevation_ft"][0:200])

    return fplatarr,fplonarr,fpaltarr

## get enroute icing data
def getIcing(_minlat,_minlon,_maxlat,_maxlon,_alt):
    _minlat = str(_minlat)
    _minlon = str(_minlon)
    _maxlat = str(_maxlat)
    _maxlon = str(_maxlon)

    icing_lat_list = []
    icing_lon_list = []
    icing_sev_list = []
```

```
    ## icing lat from db
    icing_lat_query = "SELECT Latitude FROM IcingInfo" +str(int(_alt/1000)) +" WHERE Latitude > " +
_minlat +" AND Latitude < + _maxlat + " AND Longitude >" + _minlon + " AND Longitude < "+ _maxlon
    icing_lat = pd.read_sql(icing_lat_query, icing_conn)
    for i in range(len(icing_lat)):
        icing_lat_list.append([])
        icing_lat_list[i] = icing_lat['Latitude'][i]

    ## icing lon from db
    icing_lon_query = "SELECT Longitude FROM IcingInfo" + str(int(_alt / 1000)) + " WHERE Latitude > "
+ _minlat + " AND Latitude < + _maxlat + " AND Longitude >" + _minlon + " AND Longitude < " + _maxlon
    icing_lon = pd.read_sql(icing_lon_query, icing_conn)
    for i in range(len(icing_lon)):
        icing_lon_list.append([])
        icing_lon_list[i] = icing_lon['Longitude'][i]

    ## icing sev from db
    icing_sev_query = "SELECT ICSEV FROM IcingInfo" + str(int(_alt / 1000)) + " WHERE Latitude > " +
_minlat + " AND Latitude < + _maxlat + " AND Longitude >" + _minlon + " AND Longitude < " + _maxlon
    icing_sev = pd.read_sql(icing_sev_query, icing_conn)
    for i in range(len(icing_sev)):
        icing_sev_list.append([])
        icing_sev_list[i] = icing_sev['ICSEV'][i]
    return icing_lat_list, icing_lon_list, icing_sev_list

## get nearby waypoint
def getWaypoint(_minlat,_minlon,_maxlat,_maxlon):
    _minlat = str(_minlat)
    _minlon = str(_minlon)
    _maxlat = str(_maxlat)
    _maxlon = str(_maxlon)

    wpt_name_list =[]
    wpt_lat_list = []
    wpt_lon_list = []

    ## wpt data from cifp
    wpt_name_query = "SELECT WaypointName FROM Waypoints WHERE Latitude > " + _minlat +" AND
Latitude < + _maxlat + " AND Longitude >" + _minlon + " AND Longitude < "+ _maxlon
    wpt_name = pd.read_sql(wpt_name_query,cifp_conn)
    for i in range(len(wpt_name)):
```

```
            wpt_name_list.append([])
            wpt_name_list[i] = wpt_name['WaypointName'][i]

        for i in range(len(wpt_name)):
            wpt_name_str = wpt_name_list[i]
            wpt_lat_query = "SELECT Latitude FROM Waypoints WHERE WaypointName = '"+wpt_name_str
+ "'"
            wpt_lat = pd.read_sql(wpt_lat_query,cifp_conn)
            wpt_lat_list.append([])
            wpt_lat_list[i] = wpt_lat['Latitude'][0]

            wpt_lon_query = "SELECT Longitude FROM Waypoints WHERE WaypointName = '" +
wpt_name_str + "'"
            wpt_lon = pd.read_sql(wpt_lon_query, cifp_conn)
            wpt_lon_list.append([])
            wpt_lon_list[i] = wpt_lon['Longitude'][0]
        return wpt_name_list, wpt_lat_list, wpt_lon_list
```

# Appendix F4: Obtain NAVID chart from data files

```
import gdal as gdal
import matplotlib.pyplot as plt

def getvfrSectional():
    filepath = 'New_York_VFR_Sectional/New York SEC 101.tif'
    raster = gdal.Open(filepath).ReadAsArray()
    image = plt.imshow(raster)
    plt.show()

def getvfrTerminal():
    filepath = 'New_York_TAC_99/New York TAC 99.tif'
    raster = gdal.Open(filepath).ReadAsArray()
    image = plt.imshow(raster)
    plt.show()

def getvfrterminalPlanning():
    filepath = 'New_York_TAC_99/New York TAC VFR Planning Charts 99.tif'
    raster = gdal.Open(filepath).ReadAsArray()
    image = plt.imshow(raster)
    plt.show()

def getifrlowAltitude():
    filepath = 'enr_l34/ENR_L34.tif'
    raster = gdal.Open(filepath).ReadAsArray()
    image = plt.imshow(raster)
    plt.show()

def getifrhiAltitude():
    filepath = 'enr_h12/ENR_H12.tif'
    raster = gdal.Open(filepath).ReadAsArray()
    image = plt.imshow(raster)
    plt.show()
```

# Appendix F5: Obtain elevation profile from data files

```
import urllib.request
import json
import math
import matplotlib.pyplot as plt

def haversine(lat1,lon1,lat2,lon2):
    lat1_rad=math.radians(lat1) ##math.radians: convert angle x from degrees to radians
    lat2_rad=math.radians(lat2)
    lon1_rad=math.radians(lon1)
    lon2_rad=math.radians(lon2)
    delta_lat=lat2_rad-lat1_rad
    delta_lon=lon2_rad-lon1_rad

a=math.sqrt((math.sin(delta_lat/2))**2+math.cos(lat1_rad)*math.cos(lat2_rad)*(math.sin(delta_lon/2))**2)
    d=2*6371000*math.asin(a)
    return d

def elevationData():
    with open("elevation_profile_data.json", "r") as read_file:
        js_str = json.load(read_file)

    # DISTANCE CALCULATION, mian process is obtained from the original example code
    lat_list = []
    lon_list = []
    d_list = []

    for j in range(len(js_str['results'])):
        lat_list.append(js_str['results'][j]['latitude'])
        lon_list.append(js_str['results'][j]['longitude'])

    lat0 = lat_list[-1]
    lon0 = lon_list[-1]

    for j in range(len(lat_list)):
        lat_p = lat_list[j]
        lon_p = lon_list[j]
        dp = haversine(lat0, lon0, lat_p, lon_p) / 1000    # km
        d_list.append(dp)
```

```
d_list_rev = d_list[::-1]    # reverse list

# GETTING ELEVATION
response_len = len(js_str['results'])
elev_list = []
for j in range(response_len):
        elev_list.append(js_str['results'][j]['elevation'])

return d_list_rev,elev_list
```

# Appendix G: Prototype #3

## Appendix G1: Design the map layout

```
<!DOCTYPE html>
<html>
    <head>
    <meta charset="utf-8" />
    <title>FlySmart Map</title>
    <meta name="viewport" content="initial-scale=1,maximum-scale=1,user-scalable=no" />
    <script src="https://api.mapbox.com/mapbox-gl-js/v2.2.0/mapbox-gl.js"></script>
   <link href="https://api.mapbox.com/mapbox-gl-js/v2.2.0/mapbox-gl.css" rel="stylesheet" type="text/css">
    <link href="page_style.css" rel="stylesheet" />
    </head>
<body>
        <!-- scripts for geocoder -->
    <script src="https://api.mapbox.com/mapbox-gl-js/plugins/mapbox-gl-geocoder/v4.5.1/mapbox-gl-
geocoder.min.js"></script>
    <link
    rel="stylesheet"
    href="https://api.mapbox.com/mapbox-gl-js/plugins/mapbox-gl-geocoder/v4.5.1/mapbox-gl-
geocoder.css"
    type="text/css"
    />
    <script src="https://cdn.jsdelivr.net/npm/es6-promise@4/dist/es6-promise.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/es6-promise@4/dist/es6-promise.auto.min.js"></script>
    <div id="map"></div>

    <div id="view_menu">
    <input id="nicvb/cka9ti4bj0z2q1iny1w3zqujy" type="radio" name="rtoggle" value="flight"
checked="checked"/>
    <label for="flight">Flight</label>

    <input id="mapbox-map-design/ckhqrf2tz0dt119ny6azh975y" type="radio" name="rtoggle"
value="satellite" />
    <label for="satellite">3D Satellite</label>
    </div>

    <form id="input_menu">
```

```
<!--need custom text in file upload button -> set button "adsb_upload" to replace the default file upload button-->
<input type="button" id="adsb_upload" value = "Select ADS-B file" onclick = "document.getElementById('adsb_file').click();">
<input type="file" id="adsb_file" accept=".csv" style="display:none" onclick="adsbcsv()"/>
<input type="submit" value="Update ADS-B" onclick = "upload_csv_adsb()"/>

<input type="button" id="nexradii_upload" value = "Select NEXRADII file" onclick="document.getElementById('nexrad_file').click();">
<input type="file" id="nexrad_file" accept=".csv" style="display:none" onclick="nexradcsv()"/>
<input type="submit" value="Update nexrad" onclick = "upload_csv_nexrad()"/>
</form>

<!-- map overlay menu content -->
<div class="map-overlay">

<!-- time control -->
<div class="map-overlay-inner" style ="background-color: rgba(255, 255, 255, 0.8);">
<div class="display_text">
<p>Air Traffic Database Time:</p>
<p>Weather Radar Database Time:</p>
<p>Terrain Database Time:</p>
<p>Airspace Infrastructure Time:</p>
</div>
<div class="display_text">
<p id = "adsb_time"> </p>
<p id = "nexrad_time"> </p>
<!-- <p id = "weather_radar_time">01/09/2020 21:00 UTC</p> -->
<!-- <p id = "terrain_time"> </p> -->
<p id = "terrain_time">2020-02-11 00:00 UTC</p>
<!-- <p id = "airspace_time"> </p> -->
<p id = "airspace_time">2020-05-17 12:00 UTC</p>
</div>

<hr class="solid">
<p style="font-size: 15px;">Simulation Time: <label id="slected_time">0</label> <label id='selected_time_unit'>second(s)</label></p>

<!-- <div style="background-color: #a5787894;text-align: center;"> -->
<div style="text-align: center;">
<button id="play_back" style="background-color: rgba(0, 0, 0, 0); font-size:25px; word-
```

```
wrap: break-word;border: none;" onclick = "previousFrame()">⏪</button>
                              <button id="play_auto" style="background-color: rgba(0, 0, 0, 0); font-size:25px; border: none;"onclick = "playStop()" >▶️</button>
                              <button id="play_forward" style="background-color: rgba(0, 0, 0, 0); font-size:25px; border: none;"onclick = "nextFrame()">⏩</button>
                      </div>
                      <input id="slider" type="range" min="0" max="80" step="1" value="0"/>

              </div>

              <!-- layer display control -->
              <div class="map-overlay-inner" style="border: none;">
              <fieldset>
              <button id="All" style = "font-size: 15px;background-color:#FFFFFF" onclick ="displayall()">All</button>
              <button id="RiskContours" style = "font-size: 15px;background-color:dodgerblue onclick ="displayControl('RiskContours','RiskContours')">Risk Contours</button>
              <button id="AirspaceRoute" style = "font-size: 15px; background-color:dodgerblue" onclick="displayControl('Airspace','AirspaceRoute'), displayControl('Airroute')" >Airspace/AirRoute</button>
              <button id="Landmark" style = "font-size: 15px;background-color:dodgerblue" onclick ="displayControl('Landmark','Landmark')">Landmark</button>
              <button id="NAVAID" style = "font-size: 15px;background-color:dodgerblue" onclick="displayControl('NAVAID','NAVAID')">NAVAID</button>
              <button id="Wxradar" style = "font-size: 15px;background-color:dodgerblue" onclick ="displayControl('Reflectivity','Wxradar')">Weather radar</button>
              <button id="Traffic" style = "font-size: 15px;background-color:dodgerblue" onclick ="displayControl('adsb_callsign_labels_layer','Traffic'), displayControl('adsb_velocity_labels_layer'), displayControl('adsb_altitude_labels_layer')">Traffic</button>
              <button id="Novel" style = "font-size: 15px;background-color:#FFFFFF" onclick ="display_novel_entities('Novel')">Novel Entities</button>
              </fieldset>
              </div>

       </div>

       <!-- info box for displaying aircraft performance -->
       <div id = "info_box">
              <div id="es_information"></div>
       </div>

       <!-- Function "turf" for distance calculation between points    -->
```

```
    <script src="https://npmcdn.com/@turf/turf@5.1.6/turf.min.js"></script>
    <script src="js/main_map.js"></script>

    <!-- load simulation info -->
    <script id="simulation_info" src="python_simulation/OutputDataFile/simulation_data.js"></script>
    <script src="js/simulation_info.js"></script>

    <!-- test load csv file-->
    <script src = "js/load_csv.js"></script>

    <!--load historical adsb data from csv file -->
    <script src = "js/historical_adsb.js"></script>

    <!-- load historical nexrad data from csv file-->
    <script src = "js/historical_nexrad.js"></script>

    <!--for using three.js plugins used for 3D objects -->
    <script src="https://unpkg.com/three@0.106.2/build/three.min.js"></script>
    <script src="https://unpkg.com/three@0.106.2/examples/js/loaders/GLTFLoader.js"></script>

    <!-- for using threebox.js mesh line to create 3d flight path -->
    <script src="threebox/threebox.js"></script>


</body>
</html>
```

## Appendix G2: Main execution file

```
// access token of the map designed on Mapbox studio
mapboxgl.accessToken =
'pk.eyJ1IjoibmljdmIiLCJhIjoiY2thNzBxMnl0MDAyYzJ0bmZpeW1jOHNlayJ9.p5h0jJ78qIUWcRLQ19muY
w';
var map = new mapboxgl.Map({
container: 'map',

//load our flight map at default
style: 'mapbox://styles/nicvb/cka9ti4bj0z2q1iny1w3zqujy',


zoom: 14,
center: [-73.7786925,40.6399278], //JFK airport
pitch: 60,
});

//control for switching between flight map view and satellite map view
var layerList = document.getElementById('view_menu');
var inputs = layerList.getElementsByTagName('input');

function switchLayer(layer) {
var layerId = layer.target.id;
map.setStyle('mapbox://styles/' + layerId);

}

for (var i = 0; i < inputs.length; i++) {
inputs[i].onclick = switchLayer;
}


//set default data source for airspace, airroute, and navaid
var airspaceSource = 'mapbox://nicvb.5545qn4c';
var airspaceSourceLayer = 'united_states-axvqcm';
var airrouteSource = 'mapbox://nicvb.90kqjezx';
var airrouteSourceLayer = 'a896b894_db9c_4207_82ac_8e07c5e207b4202046_1_1eto9wt.0jyd';
var navaidSource = 'mapbox://nicvb.ckaak488q0nbo2hpfm6juvwz7-2riz7';
var navaidSourceLayer = 'navaids';
```

```
var frameCount = 100;
// this is for JFK-PHL novel entity
// var currentImage = 1500037201;
var currentImage = 1500037367;


function getgifPath() {
        return (
                // 'http://localhost:8000/localhost/RiskContours/JFK-PHL novel entity/' +
                'http://localhost:8000/localhost/RiskContours/for demo only/' +

        currentImage +
        '.png'
        );
}


//slider control
var slider_ele = document.getElementById('slider');
var slider_time = slider_ele.value;

//animation control
var animationPosition = 0;
var animationTimer = false;

//geojson of adsb and novel air vehicle to    to load 3d model
var far_drone_geojson;
var adsb_geojson;
var nexrad_geojson;
var sim_geojson;

var novel_entity_label;


/////////////////////////////////////////////////////////////////////////////
// this section defined the build in map control (zoom in, zoom out, etc)

        // Add a geocoder
        map.addControl(
                new MapboxGeocoder({
                accessToken: mapboxgl.accessToken,
```

```
        mapboxgl: mapboxgl
        })
        );

    // Adds zoom and rotation controls to top right of map
    map.addControl(new mapboxgl.NavigationControl({ visualizePitch: true, showZoom: true , showCompass:
true}));


    // Adds fullscreen control to top right of map
    map.addControl(new mapboxgl.FullscreenControl({container:document.querySelector('body')}));

    // Adds scale control to top right of map
    var scale = new mapboxgl.ScaleControl({
        maxWidth: 80,
        unit: 'imperial'
    });
    map.addControl(scale);
scale.setUnit('metric');

////////////////////////////////////////////////////////////////////////////////
// this section defined all the custom data layer added to the map
function addCommonLayer()
{

    // //landmark layer
    map.addSource('landmark',{
        type: 'vector',
        url: 'mapbox://ac-0636.c46dnh3l'
    });

    map.addLayer({
        'id': 'Landmark',
        'type': 'symbol',
        'source': 'landmark',
        //'source-layer'= name of source detail
        'source-layer': '9c937979_7e9b_43d3_970e_e5b95dcdb6eb202042_1_22kxp8.7iw8s',
        'layout':
        {
            'visibility':'visible',
            'icon-image' : 'castle-15',
```

```
        'icon-size' : 1
    },
});

//////---------------------------------------------
////dummy data for nexrad csv initialization
nexrad_geojson = {
    'type': 'FeatureCollection',
    'features':[{
        'type':'Feature',
        'properties':{
            'reflectivity': ' ',
        },
        'geometry':{
            'type':'Point',
            'coordinates':[0,0]
            // 'coordinates':[-73.77,40.63]

        }
    }]
};

map.addSource('nexrad_points',{
    'type':'geojson',
    'data': nexrad_geojson
})
map.addLayer({
    'id':'Reflectivity',
    'type': 'heatmap',
    'source':'nexrad_points',
    'layout':
    {
        'visibility':'visible'
    },
    'paint':
    {
        // !!! the min and max of reflectivity value for different dataset could be different
        'heatmap-weight':[
            "interpolate",
            ["linear"],
            ['get','reflectivity'],
```

```
                -20,
                0,
                40,
                1
            ],
            // increase intensity as zoom level increases
            'heatmap-intensity': 1,
            //assign color
            'heatmap-color': [
                "interpolate",
                ["linear"],
                ["heatmap-density"],
                0,
                "rgba(0, 0, 255, 0)",
                0.1,
                "hsla(223, 98%, 42%, 0.16)",
                0.3,
                "hsla(223, 98%, 42%, 0.38)",
                0.5,
                "hsla(223, 98%, 42%, 0.54)",
                0.7,
                "hsla(229, 100%, 50%, 0.67)",
                1,
                "hsl(223, 98%, 42%)"
              ],
            // increase radius as zoom increases
            'heatmap-radius': 20,
            // decrease opacity to transition into the circle layer
            'heatmap-opacity': 0.5,

        }

});


// airspace layer
map.addSource('airspace_data',{
        type: 'vector',
        url: airspaceSource
        });
```

```
map.addLayer({
        'id': 'Airspace',
        'type': 'fill',
        'source': 'airspace_data',
        //'source-layer'= name of source detail
        'source-layer': airspaceSourceLayer,
        'layout':
        {
                'visibility':'visible'
        },
        'paint':
        {
                'fill-color': '#000000',
                'fill-opacity':["step",
                ["zoom"],
                0.33,
                10,
                0
                ]

        }
});


// air route layer
map.addSource('airroute_data',{
        type: 'vector',
        url: airrouteSource
        });

map.addLayer({
        'id': 'Airroute',
        'type': 'line',
        'source': 'airroute_data',
        'source-layer': airrouteSourceLayer,
        'layout':
        {
                'visibility':'visible'
        },
        'paint':
        {
```

```
        'line-color': '#ffff00',
        'line-opacity':0.23

    }
});


//navaid layer
map.addSource('navaid_data',{
    type: 'vector',
    url: navaidSource
    });

map.addLayer({
    'id': 'NAVAID',
    'type': 'symbol',
    'source': 'navaid_data',
    'source-layer': navaidSourceLayer,
    'layout':
    {
        'icon-image': [
            "match",
            ["get", "type"],
            ["VOR"],
            "VOR",
            ["VORTAC"],
            "VORTAC",
            ["VOR-DME"],
            "VOR-DME",
            ["NDB"],
            "NDB_the_good_one",
            ["TACAN"],
            "TACAN",
            ["NDB-DME"],
            "NDB_the_good_one",
            ["DME"],
            "VOR-DME",
            ""
            ],
        'icon-size':0.2,
        'visibility':'visible',
```

```
        'text-field':['get', 'ident'],
        'text-size': 16,
        'text-offset':[0,1.5]
    },
    'paint':
    {
        'text-color':'hsl(183, 91%, 48%)',
        'text-halo-color':'hsl(0, 0%, 0%)',
        'text-halo-width' : 1

    }
});

////dummy data for adsb csv initialization
adsb_geojson = {
    'type':'FeatureCollection',
    'features':[{
        'type': 'Feature',
        'properties':{
            'callsign': ' ',
        },
        'geometry':{
            'type': 'Point',
            'coordinates':[0, 0]
        }
    }]
};

map.addSource('adsb_labels',{
    'type':'geojson',
    'data': adsb_geojson
})
map.addLayer({
    'id': 'adsb_callsign_labels_layer',
    'type': 'symbol',
    'source':'adsb_labels',
    'layout':{
        'text-field': ['get', 'callsign'],
        'text-variable-anchor':['bottom'],
        'text-radial-offset':1.5,
        'text-justify':'center',
```

```
            'text-size':20,
            'visibility': 'visible'
        },
        'paint':{
            'text-color' : '#0F0F0F',
        }
});
map.addLayer({
        'id': 'adsb_velocity_labels_layer',
        'type': 'symbol',
        'source':'adsb_labels',
        'layout':{
            'text-field': ['get', 'velocity_l'],
            // 'text-field': ['get', 'velocity'],
            'text-variable-anchor' : ['top-left'],
            'text-radial-offset':1.5,
            'text-justify': 'left',
            'text-size':10,
            'visibility':'visible'
        },
        'paint':{
            'text-color' : '#0F0F0F',
        }
});
map.addLayer({
        'id': 'adsb_altitude_labels_layer',
        'type': 'symbol',
        'source':'adsb_labels',
        'layout':{
            'text-field': ['get', 'baroaltitude_l'],
            'text-variable-anchor' : ['top-right'],
            'text-radial-offset':1.5,
            'text-justify': 'right',
            'text-size':10,
            'visibility':'visible'
        },
        'paint':{
            'text-color' : '#0F0F0F',
        }
});
```

```
//dummy data for novel air vehicle initialization
novel_entity_label = {
	'type': 'FeatureCollection',
	'features': [
	{
	'type': 'Feature',
	'properties': {
	'description': "Novel entity",
	'icon': 'theatre'
	},
	'geometry': {
	'type': 'Point',
	'coordinates': [0,0]
	}
	}
	]
};
map.addSource('novel_entity_labels',{
	'type':'geojson',
	'data': novel_entity_label
})
map.addLayer({
	'id': 'novel_entity_labels_layer',
	'type': 'symbol',
	'source': 'novel_entity_labels',
	'layout': {
	'text-field': 'Novel air vehicle',
	// 'text-variable-anchor': ['top', 'bottom', 'left', 'right'],
	'text-font': ['Open Sans Semibold', 'Arial Unicode MS Bold'],
	'text-variable-anchor': ['bottom'],
	'text-radial-offset': 1.5,
	'text-justify': 'auto',
	// 'text-justify': 'center',
	'text-size': 30, // to adjust the callsign text size
	'visibility':'visible'
	},
	paint: {
		"text-color": "#483D8B"
	  }
	});
```

```
	// load novel air vehicle info by calling function in another js file
	load_simulation_data();

	// load adsb data by calling function in another js file
	load_adsb();

	// load weather radar data by calling function in another js file
	load_nexrad();


}

function addRiskLayer()
{
	//risk contours
	map.addSource('risk_contours',
	{
		type: 'image',
		url: getgifPath(),
		coordinates: [
		//KJFK - KPHL
		[-76.2526, 41.2500],	//ulc
		[-71.8820, 41.2500], //urc ----
		[-71.8820, 39.2353], //lrc
		[-76.2526, 39.2353] //llc ----
		//KJFK - KLGA
		// [-74.4113, 40.9142],	//ulc
		// [-73.4819, 40.9142], //urc ----
		// [-73.4819, 40.410867], //lrc
		// [-74.4113, 40.410867] //llc ----
		]
	});
	map.addLayer({
		id: 'RiskContours',
		'type': 'raster',
		'source': 'risk_contours',
		'layout':{
			'visibility':'visible'
		},
		'paint': {
```

```
        'raster-fade-duration': 0,
        "raster-opacity" : 0.5 // to adjust the risk countour image opacity
        }
        });
}


function add3dTerrainLayer()
{
    map.addSource('mapbox-dem', {
        'type': 'raster-dem',
        'url': 'mapbox://mapbox.mapbox-terrain-dem-v1',
        'tileSize': 512,
        'maxzoom': 14
        });
    //      // add the DEM source as a terrain layer with exaggerated height
    map.setTerrain({ 'source': 'mapbox-dem', 'exaggeration':1});
}


/////////////////////////////////////////////////////////////////////////////////////////
// this section is for changing the visibility of the data layer
//load all the data layer when the map style first load
map.on('load', function()
{
    addCommonLayer();


});

//reload all the data layer when the map style change
//Ref: https://bl.ocks.org/tristen/0c0ed34e210a04e89984
map.on('style.load',function()
{
    addCommonLayer();

    // check which map style is the current selected one
    var flight_map_checked = document.getElementById("nicvb/cka9ti4bj0z2q1iny1w3zqujy").checked;
    var satellite_map_checked = document.getElementById("mapbox-map-design/ckhqrf2tz0dt119ny6azh975y").checked;
```

```
        //// load extra layer to the selected map style
        if (flight_map_checked == true)
        {
                console.log("Current map style: Flight map, added risk image layer")
                addRiskLayer();

        }
        if(satellite_map_checked == true)
        {
                console.log("Current map style: 3D satellite map, add 3d terrain layer")
                add3dTerrainLayer();
        }


});

//use in all dispaly button, pass in layer name and button id
function displayControl(layername, btn)
{
        var visibility = map.getLayoutProperty(layername,'visibility');
        var click_btn = document.getElementById(btn);
        // console.log(layername,':    ',visibility)

        if (visibility == 'visible')
        {
                map.setLayoutProperty(layername, 'visibility', 'none');
                if (click_btn != null)
                {
                    click_btn.style.backgroundColor = "#FFFFFF";
                }
        }
        else
        {
                map.setLayoutProperty(layername, 'visibility', 'visible');
                if (click_btn != null)
                {
                        click_btn.style.backgroundColor = "#1E90FF";
                }
        }

}
```

```
function displayall()
{
	var all_btn = document.getElementById('All');
	if (all_btn.style.backgroundColor == "dodgerblue")
	{
		all_btn.style.backgroundColor = '#FFFFFF';
	}
	else
	{
		all_btn.style.backgroundColor = '#1E90FF';
	}

	displayControl('Icing','Wxradar');
	displayControl('Landmark','Landmark');
	displayControl('Airspace','AirspaceRoute');
	displayControl('Airroute');
	displayControl('NAVAID','NAVAID');
	displayControl('RiskContours','RiskContours');
	displayControl('drone_labels_layer','Traffic');
	displayControl('drone_labels_spd_layer');
	displayControl('drone_labels_alt_layer');
	// display_novel_entities('Novel')
}

function display_novel_entities(btn)
{
	var click_btn = document.getElementById(btn);
	var visibility = map.getLayoutProperty('novel_entity_labels_layer','visibility');
	// console.log('layer visibility: ', visibility);

	if (novel_entity_status == 1)
	{
		novel_entity_status = 0
		click_btn.style.backgroundColor = "#FFFFFF";
		map.setLayoutProperty('novel_entity_labels_layer','visibility','none');

	}
	else
	{
```

```
            novel_entity_status = 1
            click_btn.style.backgroundColor = "#1E90FF";
            map.setLayoutProperty('novel_entity_labels_layer','visibility','visible');
        }

}

////////////////////////////////////////////////////////////////////////////////
// this section is for the media control
function stop() {

    document.getElementById("play_auto").textContent = "▶️";
    console.log("stop");
    if(animationTimer)
    {
        clearTimeout(animationTimer);
        animationTimer = false;
        return true;
    }
    // return false;
}

function play()
{
    document.getElementById("play_auto").textContent = "⏸️";
    if (slider_time < slider_ele.max)
    {
        slider_time += 1;
        slider_ele.value = String(slider_time);
    }
    else
    {
        slider_time = 0;
        slider_ele.value = String(slider_time);
    }
    // Main animation driver. Run this function every 5s
    animationTimer = setTimeout(play, 2000);
    console.log("play");
}

function playStop()
```

```
{
	if (!stop()) {
		play();
	}
}

function nextFrame()
{
	stop();
	slider_time += 5;
	slider_ele.value = String(slider_time);
	console.log("next");
}

function previousFrame()
{
	stop();
	slider_time -= 5;
	slider_ele.value = String(slider_time);
	console.log("previous");
}
```

# Appendix G3: Convert data in CSV file to arrays

```
var adsb_csv;
var nexrad_csv;

// convert data in csv file to an array
// ref: https://sebhastian.com/javascript-csv-to-array/
function csvToArray(str, delimiter = ",")
{
    // slice from start of text to the first \n index
  // use split to create an array from string by delimiter
    const headers = str.slice(0, str.indexOf("\n")).split(delimiter);

  // slice from \n index + 1 to the end of the text
  // use split to create an array of each csv value row
    const rows = str.slice(str.indexOf("\n") + 1).split("\n");

  // Map the rows
  // split values from each row into an array
  // use headers.reduce to create an object
  // object properties derived from headers:values
  // the object passed as an element of the array
  const arr = rows.map(function (row) {
        const values = row.split(delimiter);
        const el = headers.reduce(function (object, header, index) {
            object[header] = values[index];
            return object;
        }, {});
        return el;
    });

  // return the array
    return arr;
}


function upload_csv_adsb()
{
  const myForm = document.getElementById("input_menu");
  const adsb_file = document.getElementById("adsb_file");
```

```
    // prevent browser from executing the default action of the selected element
    myForm.addEventListener("submit", function(e)
    {
       e.preventDefault();
    });

    //// load adsb data from csv file
    // .files[0] : return to the file object at the index 0
    const file1 = adsb_file.files[0];
    const csvreader1 = new FileReader();
    csvreader1.readAsText(file1);
    csvreader1.onload = function()
    {
          adsb_csv = csvToArray(csvreader1.result);

          //get adsb start time of this csv file
          adsb_t=get_adsb_start_time(adsb_csv);
          console.log('adsb_start time:',adsb_t)
          var adsb_date = convertTimestamp(adsb_t)
          document.getElementById("adsb_time").innerHTML = adsb_date;

    }

    //// call function in historical_adsb.js to get adsb data
    get_adsb_data();

}

function upload_csv_nexrad()
{
    const myForm = document.getElementById("input_menu");
    const nexrad_file = document.getElementById("nexrad_file");
    // prevent browser from executing the default action of the selected element
    myForm.addEventListener("submit", function(e)
    {
       e.preventDefault();
    });

    //// load nexrad data from csv file
    // // .files[0] : return to the file object at the index 0
    const file2 = nexrad_file.files[0];
```

```
    const csvreader2 = new FileReader();
    csvreader2.readAsText(file2);
    csvreader2.onload = function()
    {
        nexrad_csv = csvToArray(csvreader2.result);
        nexrad_date = convertTimestamp(nexrad_csv[0].Time)
        // nexrad_date = convertNEXRADTime(nexrad_csv[0].Time)
        // console.log(nexrad_date)
        // console.log(nexrad_csv[0].sweepTime)
        document.getElementById("nexrad_time").innerHTML = nexrad_date;
    }

    ////initialize nexrad display
    get_nexrad_data();
}


//// function link to select adsb file button in .html
function adsbcsv()
{
    console.log("adsb_csv");
    if (adsb_csv != null)
    {
        console.log('new adsb file will be upload');

        //clear current adsb data
        adsb_csv=[];
        adsb_data_t = [];
        adsb_loc_t = [];
        ////get new adsb start time of this csv file
        // adsb_t=get_start_time_from_csv(adsb_csv);
        // get_adsb_data();
        // console.log('prepare for new csv file: ',adsb_csv);
    }
    else
    {
        console.log('first adsb file select');
    }

}
```

```
//// function link to select nexrad file button in .html
function nexradcsv()
{
   console.log('nexrad_csv');
   if (nexrad_csv != null)
   {
      console.log('new nexradii file will be upload');
      ////clear current nexrad data
      nexrad_csv=[];
   }
   else
   {
      console.log('first nexrad file select');
   }

}

function convertTimestamp(timestamp) {
   var date = new Date(timestamp * 1000), // Convert the passed timestamp to milliseconds
         yyyy = date.getUTCFullYear(),
         mm = ('0' + (date.getUTCMonth() + 1)).slice(-2),    // Months are zero based. Add leading 0.
         dd = ('0' + date.getUTCDate()).slice(-2),                // Add leading 0.
         hh = date.getUTCHours(),
         min = ('0' + date.getUTCMinutes()).slice(-2),         // Add leading 0.
         time;

   time = yyyy + '-' + mm + '-' + dd + ' ' + hh + ':' + min + ' UTC';
   return time;
}
```

# Appendix G4: Mapping ADS-B data for display

```
var adsb_t =0;
var adsb_data_t = []
var adsb_loc_t = []
var slider_time_diff;
var prev_slider_time = 0;


// recursive function monitoring incoming adsb file
function load_adsb()
{
	update_adsb = setInterval(function()
	{
		update_adsb_loc();
		get_adsb_data();
	},1000);
}


//get start time from adsb csv file
// function use in load_csv.js
function get_adsb_start_time (csv_obj)
{
	var values = []
	// for (var i =0; i< 1000; i ++)
	for (var i =0; i< csv_obj.length-1; i ++)
	{
		// console.log(csv_obj[i].time)
		temp_time = parseInt(csv_obj[i].time)
		// console.log(temp_time)
		if (isNaN(temp_time) == false)
		{
			values.push(parseInt(csv_obj[i].time));
		}
	}
	return min = Math.min.apply(null,values)
}

// get adsb data that match the time in simulation
```

```
function get_adsb_data()
{
	adsb_data_t = []
	for (var i =0; i< adsb_csv.length-1;i++)
	{
		if (parseInt(adsb_csv[i].time) == adsb_t)
		{
			adsb_data_t.push({
				time: parseInt(adsb_csv[i].time),
				icao : adsb_csv[i].icao24,
				latitude: parseFloat(adsb_csv[i].lat),
				longitude: parseFloat(adsb_csv[i].lon),
				velocity: parseFloat(adsb_csv[i].velocity),
				heading: parseFloat(adsb_csv[i].heading),
				vertrate: parseFloat(adsb_csv[i].vertrate),
				callsign: adsb_csv[i].callsign,
				onground : (adsb_csv[i].onground).toLowerCase(),
				baroaltitude: adsb_csv[i].baroaltitude,
				geoaltitude: adsb_csv[i].geoaltitude,
				lastposupdate: parseInt(adsb_csv[i].lastposupdate),
			});

		}
	}
	// draw adsb on map
	plot_air_traffic();
}

// this function is used to draw adsb on map
function plot_air_traffic()
{
	// store adsb position info at time t in a list
	adsb_loc_t =[];
	for (var i=0; i<adsb_data_t.length;i++)
	{
		//filter adsb traffic by onground status, keep the flying ones only
		if(adsb_data_t[i].onground == "false")
		{
			// create an object for each aircraft that matches the condition, set their properties
			adsb_loc_t.push({
				type:'Feature',
```

```
                geometry:
                {
                    type: 'Point',
                    coordinates:[Number(adsb_data_t[i].longitude), Number(adsb_data_t[i].latitude)]
                },
                properties:
                {
                    heading:Number(adsb_data_t[i].heading),
                    callsign: adsb_data_t[i].callsign,
                    icao: adsb_data_t[i].icao,
                    velocity: Number(Math.round(adsb_data_t[i].velocity)),
                    baroaltitude: Number(Math.round(adsb_data_t[i].baroaltitude)),
                    // for label display
                    velocity_l: Math.round(adsb_data_t[i].velocity)+ 'kts',
                    baroaltitude_l: Math.round(adsb_data_t[i].baroaltitude) + 'm',

                }
            });
        }

    }

    // find the adsb data layer by using 'sourceId', update the data layer
    adsb_geojson = {
        'type': 'FeatureCollection',
        features:adsb_loc_t,
    };
    map.getSource('adsb_labels').setData(adsb_geojson);
    var satellite_map_checked = document.getElementById("mapbox-map-
design/ckhqrf2tz0dt119ny6azh975y").checked;
    if(satellite_map_checked == true)
    {
        map.setPaintProperty('adsb_callsign_labels_layer', 'text-color', '#ffffff')
        map.setPaintProperty('adsb_velocity_labels_layer', 'text-color', '#ffffff')
        map.setPaintProperty('adsb_altitude_labels_layer', 'text-color', '#ffffff')
    }

    // add 3d model to map
    for (var k =0; k<adsb_loc_t.length;k++)
    {
        adsb_3d_model(k);
```

```
	}
}

// this function is define to match the adsb display with the media control
function update_adsb_loc()
{

	//prev_slider_time != (current) slider_time -> there is change on slider input OR the in initial state (prev_slider_time =0)
	if (prev_slider_time != slider_time)
	{

		//playback (drag slider backward)
		if (prev_slider_time>slider_time)
		{
			//difference between previous slider time and current slider time
			slider_time_diff = (prev_slider_time-slider_time);

			//match adsb_time to the current slider time to load adsb data
			adsb_t -= slider_time_diff;

		}
		//(drag slider forward)
		else //prev_slder_time < slider_time
		{
			//difference between previous slider time and current slider time
			slider_time_diff = (slider_time-prev_slider_time);

			//match adsb_time to the current slider time to load adsb data
			adsb_t += slider_time_diff;
		}

		//load/plot adsb traffic on the map
		get_adsb_data();
		for (var k =0; k<adsb_loc_t.length;k++)
		{
			adsb_3d_model(k);
		}

		//update prev_slider_time
		prev_slider_time = slider_time;
```

```
	}
}

// this function is used for plotting a 3d aircraft model for each aircraft in the adsb data
function adsb_3d_model(i)
{
	// three.js 3D object variables
	// parameters to ensure the model is georeferenced correctly on the map
	modelOrigin = adsb_loc_t[i].geometry.coordinates;
	modelAltitude = adsb_loc_t[i].properties.baroaltitude;

	//convert heading, from angle to radians
	var heading_rad = parseFloat(adsb_loc_t[i].properties.heading)*(Math.PI/180);
	var modelRotate = [-1.5,heading_rad,3];

	var modelAsMercatorCoordinate = mapboxgl.MercatorCoordinate.fromLngLat(
		modelOrigin,
		modelAltitude
	);

	// transformation parameters to position, rotate and scale the 3D model onto the map
	var modelTransform = {
		translateX: modelAsMercatorCoordinate.x,
		translateY: modelAsMercatorCoordinate.y,
		translateZ: modelAsMercatorCoordinate.z,
		rotateX: modelRotate[0],
		rotateY: modelRotate[1],
		rotateZ: modelRotate[2],
		scale: modelAsMercatorCoordinate.meterInMercatorCoordinateUnits()
	};

	var THREE = window.THREE;
	var layerid = "Adsb3d"+ String(i);
	// configuration of the custom layer for a 3D model per the CustomLayerInterface
	var customLayer = {
		id: layerid,
		type: 'custom',
		renderingMode: '3d',
		onAdd: function (map, gl) {
			this.camera = new THREE.Camera();
			this.scene = new THREE.Scene();
```

```
        var directionalLight2 = new THREE.DirectionalLight(0xffffff);
        directionalLight2.position.set(0, 70, 100).normalize();
        this.scene.add(directionalLight2);

        // use the three.js GLTF loader to add the 3D model to the three.js scene
        var loader = new THREE.GLTFLoader();
        loader.load(
            'http://localhost:8000/localhost/PlaneModel/hl64/scene.gltf',
            function (gltf) {
                this.scene.add(gltf.scene);
            }.bind(this)
        );
        this.map = map;

        // use the Mapbox GL JS map canvas for three.js
        this.renderer = new THREE.WebGLRenderer({
            canvas: map.getCanvas(),
            context: gl,
            antialias: true
        });

        this.renderer.autoClear = false;
    },
    render: function (gl, matrix) {
        var rotationX = new THREE.Matrix4().makeRotationAxis(
            new THREE.Vector3(1.2, 0, 0),
            modelTransform.rotateX
        );
        var rotationY = new THREE.Matrix4().makeRotationAxis(
            new THREE.Vector3(0, 1.2, 0),
            modelTransform.rotateY
        );
        var rotationZ = new THREE.Matrix4().makeRotationAxis(
            new THREE.Vector3(0, 0, 1.2),
            modelTransform.rotateZ
        );

        var m = new THREE.Matrix4().fromArray(matrix);
        var l = new THREE.Matrix4()
            .makeTranslation(
```

```
                        modelTransform.translateX,
                        modelTransform.translateY,
                        modelTransform.translateZ
                    )
                    .scale(
                        new THREE.Vector3(
                            modelTransform.scale,
                            -modelTransform.scale,
                            modelTransform.scale
                        )
                    )
                    .multiply(rotationX)
                    .multiply(rotationY)
                    .multiply(rotationZ);

                this.camera.projectionMatrix = m.multiply(l);
                this.renderer.state.reset();
                this.renderer.render(this.scene, this.camera);
                this.map.triggerRepaint();
            }
        };
        var map_layer = map.getLayer(layerid);
        if (prev_slider_time != slider_time)
        {
            if (typeof map_layer == 'undefined')
            {
                map.addLayer(customLayer, 'waterway-label');
            }
            else
            {
                map.removeLayer(layerid);
                map.addLayer(customLayer, 'waterway-label');
            }
        }


    }
```

# Appendix G5: Mapping NexRad data for display

```
var nexrad_t = 0;
var nexrad_data_t =[]
var nexrad_loc_t = []

// recursive function monitoring incoming weather radar file
function load_nexrad()
{
	update_nexrad = setInterval(function()
	{
		get_nexrad_data();
	},2000);

}


// push current nexrad data read from csv
function get_nexrad_data()
{
	nexrad_data_t = []

	for (var i = 0; i< nexrad_csv.length-1;i++)
	{
		nexrad_data_t.push({
			time: nexrad_csv[i].sweepTime,
			reflectivity: nexrad_csv[i].value,
			latitude: parseFloat(nexrad_csv[i].Latitude),
			longitude: parseFloat(nexrad_csv[i].Longitude),
			heightRel : parseFloat(nexrad_csv[i].HeightRel),
			heightASL: parseFloat(nexrad_csv[i].HeightASL),
		});
	}
	plot_nexrad_data();
}


//plot nexrad weather data
function plot_nexrad_data()
{
```

```
    // create geojson object for the nexrad weather data point
    nexrad_loc_t = [];
    for (var i =0; i < nexrad_data_t.length; i ++)
    {
        nexrad_loc_t.push({
            type:'Feature',
            geometry:
            {
                type:'Point',
                coordinates:[Number(nexrad_data_t[i].longitude),Number(nexrad_data_t[i].latitude)]
            },
            properties:
            {
                reflectivity: nexrad_data_t[i].reflectivity,
            }
        });
    }

    //find the weather radar data layer by using 'sourceId', update the data layer
    nexrad_geojson = {
        'type' : 'FeatureCollection',
        features:nexrad_loc_t,
    };
    map.getSource('nexrad_points').setData(nexrad_geojson);
}
```